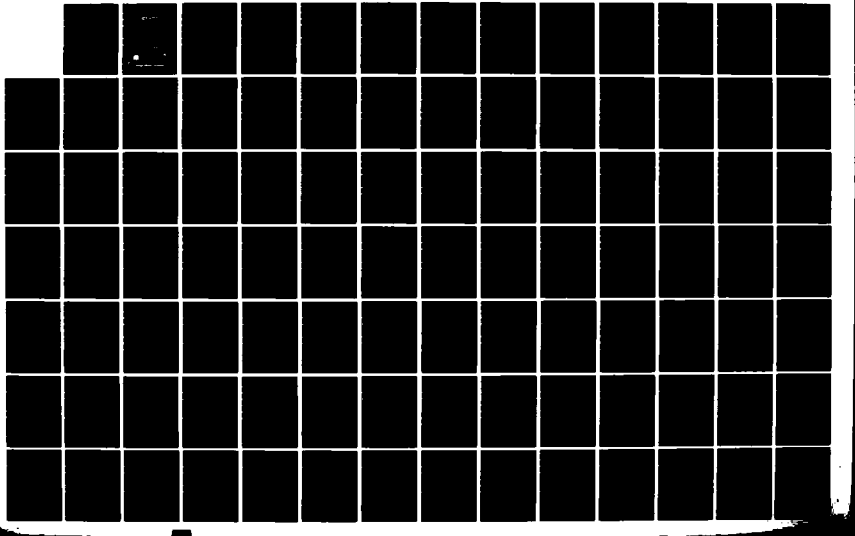
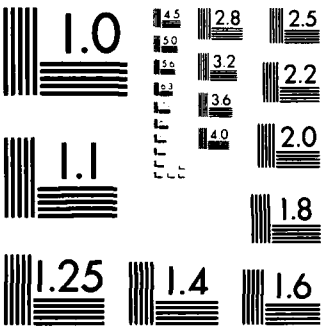
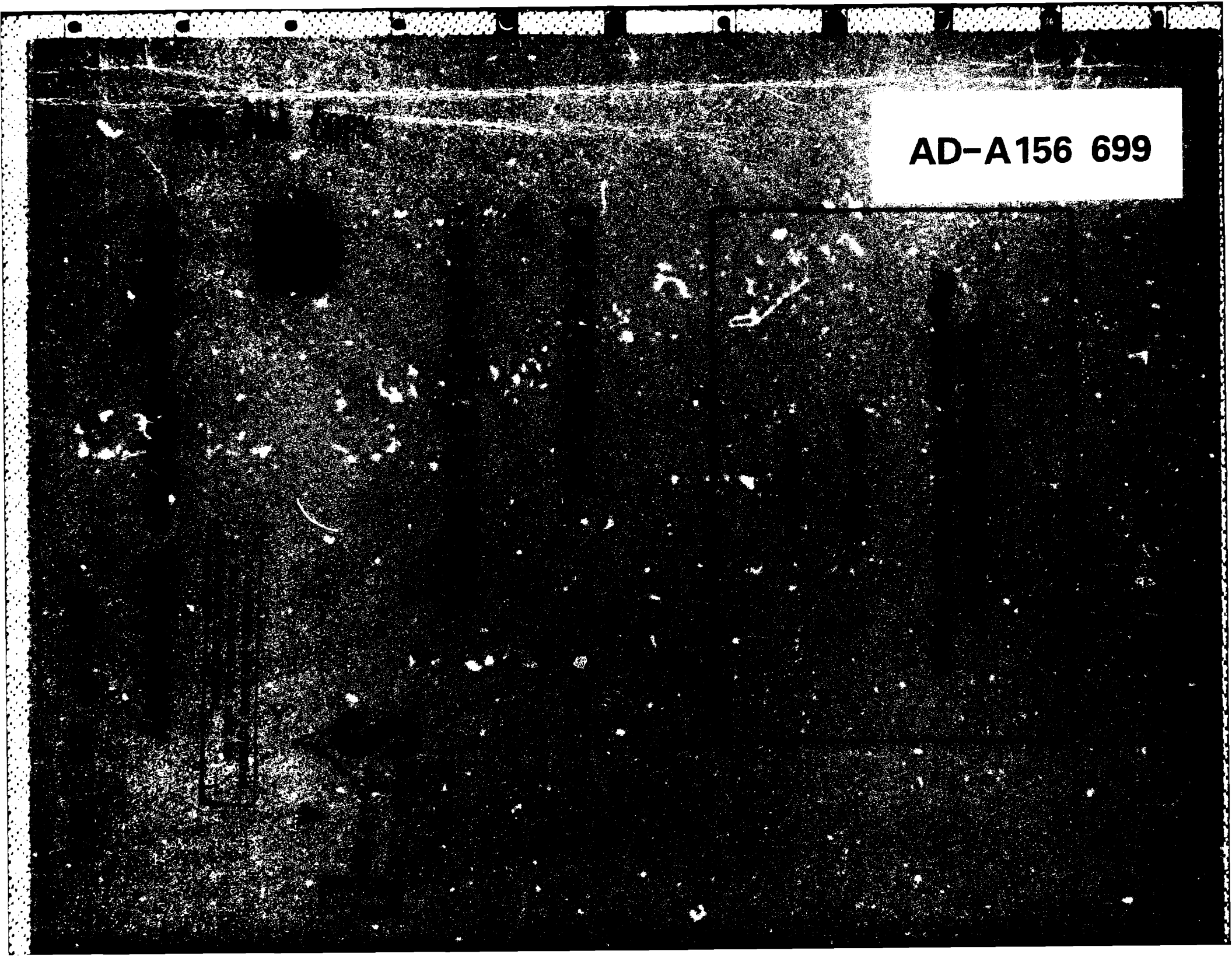
AD-A156 699 IMPLEMENTATION ISSUES FOR ALGORITHMIC VLSI (VERY LARGE SCALE INTEGRATION)..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE A L FISHER OCT 84 1/2

UNCLASSIFIED F33615-81-K-1539 F/G 9/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A



Implementation Issues for Algorithmic VLSI Processor Arrays

Allan L. Fisher

October, 1984

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy.

This research was supported by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539; by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659; by a National Science Foundation graduate fellowship; and by an IBM graduate fellowship.

Abstract

Advances in very large scale integration (VLSI) have led to a great deal of interest in highly parallel cellular architectures as high-performance solutions to computational bottlenecks. These architectures pose a number of novel implementation issues not encountered in traditional designs. This thesis investigates and resolves some of the most important of these issues, taking both practical and theoretical points of view.

Chapter 2 considers the subject of programmability for arrays of processors. It describes some of the advantages and disadvantages of flexible implementations, and discusses processor design issues in light of the requirements of the array environment. In particular, it describes the design of the PSC, a chip designed for systolic array implementations, which has been realized in nMOS. It also discusses alternative combinations of processors and memories for the implementation of systolic algorithms. Chapter 3 examines the question of synchronization in large arrays of processors. It gives asymptotic upper and lower bounds for the speed of clocked arrays of differing topologies under differing models of clock skew, and also makes some observations on practical issues of synchronization in large systems. Chapter 4 concentrates on implementations in which arithmetic is broken up into serial bit or subword steps; it shows how systolic serialized systems can be designed, and discusses some practical issues of cost and performance. Finally, Chapter 5 presents a case study of architectures for a family of information retrieval tasks; in addition to presenting a new, efficient architecture, it discusses many of the important but usually unquantified practical factors which impinge on system-building decisions.

Acknowledgments

It is a pleasure to thank the members of my thesis committee, Roberto Bisiani, Raj Reddy and James Wheeler, for their advice and encouragement. It is a particular pleasure to thank H. T. Kung, my thesis advisor, who has been a constant source of wisdom, energy, and inspiration. This thesis would also not have been possible without the intellectual stimulation provided by my friends and colleagues and the outstanding facilities provided by the technical staff at CMU CSD. I am also grateful to the National Science Foundation and IBM Corporation for their support in the form of graduate fellowships. Finally, my thanks go to my wife, Eden, who makes it all worthwhile.

Table of Contents

List of Figures

List of Tables

1 Introduction

Since the late 1970's, dramatic increases in integrated circuit density have reawakened interest in the construction of large cellular processor arrays. Such arrays had received theoretical and practical attention much earlier [21, 39], but contemporary technology was incapable of producing truly powerful arrays at reasonable cost. Another result of the high cost of hardware was that little practical attention was paid to very special-purpose machines, although it is for just such machines that it is easiest to make use of massive parallelism.

The apparent practicality of large array machines raises a number of implementation issues which vary in significant ways from their counterparts in traditional architectures. Among these are programming, synchronization, bit-serial and other "sliced" implementations, and questions of modularity and processing/memory tradeoffs. The four main chapters of this thesis explore and resolve some of these issues from both practical and theoretical viewpoints. The point of departure for much of this work is the study of systolic arrays, discussed below; however, many of the results apply equally to any cellular VLSI architecture. The common thread running through this work is the attempt to understand and solve implementation problems, which are usually approached *ad hoc*, in a structured and general way. This goal is more feasible for VLSI processor arrays than for computer architecture in general because of the arrays' scalability and regularity.

Systolic arrays Beginning in 1978 [20, 29], *systolic algorithms* [26] have been proposed as means of achieving high performance in the solution of a large family of computation-intensive problems. As sketched in Figure 1-1, the idea behind systolic algorithms is to use each input value many times, passing it through a simple and

regular network of processing elements, or cells, each of which performs some relatively simple computation for each value. Although the figure shows a one-dimensional pipeline, two-dimensional grids and trees are also used. For applications which admit such structures, the systolic approach can yield speedups relative to I/O bandwidth which are proportional to the number of processors used. In fact, a great many algorithms based on such structures have been devised [14].

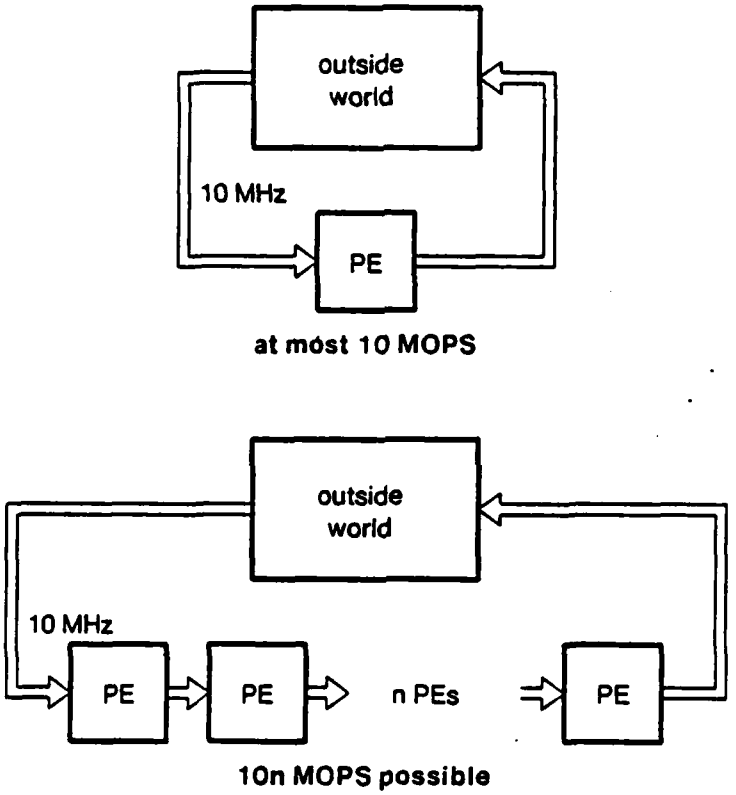


Figure 1-1: Principle of systolic algorithms.

The possibility of large speedups with relatively modest I/O bandwidth is just one of a number of advantages of the systolic approach. Other advantages, such as modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global fan-in and fan-out, and (possibly) fast response time, are also characteristic [26].

Programmability Since systolic algorithms generally consist of a few types of simple processors, or systolic cells, connected in a regular pattern, they are less expensive to design and implement than more general machines. This advantage is offset by the fact that a particular systolic system can generally be used only on a narrow set of problems, and thus design cost cannot be amortized over a large number of units. One way to approach this problem is to provide a programmable means of implementation for many systolic algorithms.

The systolic environment, by virtue of its emphasis on continuous, regular flow of data and fairly simple per-cell processing, imposes new design requirements for programmable processors which are quite different from those found in a general-purpose system. Chapter 2 considers these issues from two points of view. Section 2.1 presents the architecture of the programmable systolic chip (PSC), a single-chip microprocessor suitable for use in groups of tens or hundreds for the efficient implementation of a broad variety of systolic arrays. The processor has been fabricated in nMOS, and has been integrated into a demonstration system. The PSC architecture is inherently five to ten times more efficient for typical systolic implementations than are standard microprocessors.

Section 2.2 describes and analyzes structures that differ from the PSC approach in that processors and memory elements do not exhibit the same one-to-one correspondence found in most discussions of systolic algorithms. One approach is the inclusion of local memory in processing elements to save hardware or decrease I/O bandwidth; Section 2.2.1 makes some practical observations on modularity issues and discusses I/O-optimal multiplexing for hardware savings. The other approach is the complete separation of data from computation, which yields some advantages in modularity and flexibility; Section 2.2.2 discusses a hardware structure for the efficient implementation of data access and movement.

Synchronization In order for the elements of a VLSI processor array to communicate among themselves, some provision must be made for synchronization of data transfer. The simplest means of synchronization is the use of a global clock. Unfortunately, large clocked systems can be difficult to implement because of the inevitable problem of clock skews and delays, which can be especially acute in VLSI systems as feature sizes shrink. For the near term, good engineering and technology

improvements can be expected to maintain the feasibility of clocking in such systems; however, clock distribution problems crop up in any technology as systems grow. An alternative means of enforcing necessary synchronization is the use of self-timed, asynchronous schemes, at the cost of increased design complexity and hardware cost.

Chapter 3 gives simple and natural mathematical models for the time required to clock large arrays of communicating cells. Depending on physical circumstances, appropriate variations of the models may be applied. For each variant model, we derive upper and lower bounds for clock skew in arrays of different topologies. For example, we show that if clock skew between two points is related to the difference of their clock distribution delays, arrays may be clocked in constant time, independent of their size. On the other hand, if clock skew is related to the sum of the distribution delays, an $n \times n$ array will need a clock period which grows with n. We also suggest some new clocking schemes, and make some observations on their practicality and the applicability of the variant skew models.

Serialized implementations Serialized implementations of arithmetic operations offer advantages of simple, repetitive designs and high clock speeds. Balancing these advantages are questions of the time and space overhead of extra latches, the need for applications to provide higher degrees of parallelism, and bounds on clock speed other than combinational delay. Chapter 4 discusses two aspects of such designs. First, Section 4.1 clarifies the issues of serialization and pipelining involved in the design of such arrays, and shows how serialized systolic designs may be derived from word-parallel designs. As an example, we give derivations of some previously published bit-serial convolvers. Section 4.2 then discusses some of the cost/performance/designability tradeoffs of serialized designs, and discusses their implications in some specific cases.

Dictionary machines — a case study Very often, the choice of the best structure for a special purpose system will depend on such non-analytical or "constant factor" issues as modularity or technology properties. Chapter 5 illustrates some of these issues in the context of dictionary machines. The chapter first considers a number of tree-structured VLSI multiprocessor designs which have been proposed for performing a group of dictionary operations (INSERT, DELETE, EXTRACTMIN, NEAR, etc.) on

a set of keys. These designs typically use one processor for each key stored and operate with constant throughput, assuming unit time to communicate and compare keys. This assumption breaks down in applications with long keys. The chapter presents a new trie-based design which uses a number of processors proportional to the maximum length of a key to achieve constant throughput, regardless of key length. The design has important practical advantages over the family of tree-structured machines, and demonstrates that processor-intensive VLSI structures are not always the best route to a high-performance system.

2 Programmability

Because of their regularity of structure and simplicity of basic components, systolic arrays are inexpensive to design and implement, in comparison to other structures of equal performance. This advantage is offset, however, by the fact that a given systolic algorithm is usually tailored to a particular application, and hence development costs cannot be spread over the large number of units typical of a general-purpose processor. It is therefore appropriate to consider a range of points along a generality-performance tradeoff curve. Applications with stringent throughput demands, such as many signal processing tasks, will often justify full custom design, while other applications may be handled by an existing, more flexible design. Existing systolic array implementations span this spectrum. Early test implementations [18, 30] were full custom single-purpose devices, as is the GEC correlator chip [12]. In an intermediate range of flexibility are the ESL systolic processor [7, 50] and a forthcoming ESL systolic chipset for floating-point matrix computations, both of which are programmable for a range of signal processing tasks. At the very general end of the spectrum is the NOSC systolic array testbed [9, 47, 48], which is assembled from general-purpose microprocessors and provides both one- and two-dimensional array communication structures. Another highly flexible approach, used in the PSC project [15], is the design of widely applicable building blocks.

In order not to incorporate the overhead of chip design, fabrication, testing, and quality assurance into each construction of a systolic machine, and yet to achieve reasonable performance and chip counts, it is useful to have some flexible means of assembling many different types and sizes of systolic arrays from a small number of

building-blocks. Such a tool, in order to support a wide variety of such algorithms, must provide programmability both within individual cells and at the interconnection level. Note that this configurability need only represent "design-time" or "compile-time" rather than "run-time" flexibility; once built and programmed, a systolic array is typically expected to perform the same computation many times.

Section 2.1 of this chapter describes the goals and architecture of the CMU programmable systolic chip, a particular example of such an implementation. Some good and bad features of the architecture are discussed, and some general observations on programmability are made. Section 2.2 then discusses two different approaches to programmable implementations of systolic arrays. The first is the use of sizable local memories to reduce I/O bandwidth requirements or to trade off hardware cost and performance; we outline the design issues for such schemes and provide new observations on modularity or lack thereof in these designs. The second approach involves the decoupling of the flow of data from processing; we provide a scheme that allows mostly serial access to data, making implementations with serial technologies or with only local communication over large memories possible.

2.1. The Programmable Systolic Chip

The PSC project [15, 16] was stimulated by the need for a convenient means of realizing systolic algorithms. The PSC design attempts to provide a basis for the efficient implementation of such algorithms, and succeeds reasonably well. Its structure is intrinsically much more efficient for this task than a general-purpose microprocessor. Despite very general functionality, the architecture imposes relatively little performance overhead with respect to the arithmetic operations performed. This section describes the PSC and evaluates its effectiveness.

2.1.1. Structure of the PSC

The primary goal of the PSC architecture was to support reasonably efficient implementation of a very broad variety of systolic algorithms. To this end, a set of *target applications* was chosen. An obvious candidate was the field of signal and image processing; problems in this area often demand real-time solution and many systolic algorithms for such problems have already been designed [24, 29]. Another choice was error detection and correction coding, Reed-Solomon coding [36, 42] in particular. In addition, sort/merge of large files was chosen as a representative of data-processing applications.

Given these goals, the following design issues were considered:

- Locality and flexibility of control: For some systolic arrays, including many signal processing and matrix arithmetic algorithms, simple global control would be sufficient. However, many other algorithms require different actions in different phases (e. g., loading of coefficients), as well as data-dependent actions within each systolic cycle.

- Chip count: Keeping a systolic cell on a single chip has two advantages. First, it allows the functional blocks of the processor to operate together without paying the time and pinout penalty of off-chip communication. Second, it allows systolic arrays to be constructed with small chip count. The possibilities for putting *more* than one cell on a chip seem limited for the near future; given fairly complex processors, advances in miniaturization will probably be more usefully spent on increased word size and functional capability. This situation obviates the need for on-chip configurability—system configuration (or reconfiguration) is done at the board level.

- Primitive operations: The primitive arithmetic, logical and control operations which a processor can perform are critical to its efficiency. In particular, fast multiplication is needed for the effective implementation of most signal and image processing algorithms. Provisions for multiple-precision arithmetic can extend a processor's utility to problems requiring larger word size, without heavy performance penalties.

- Intercell communication: A principal feature of systolic arrays is the continuous flow of data between cells. Efficient implementation of such arrays requires wide I/O ports and data paths. Provision must also be made for the transmission of pipelined systolic control signals.

- Internal parallelism: Partition of a processor's function into units which can operate in parallel enhances performance.

- Control structure: Horizontally microprogrammed control structures provide flexibility in programming new applications and promote parallelism within a processor. The usual drawback of horizontally microprogrammed architectures, the difficulty of exploiting that parallelism, is eased by the fact that systolic algorithms usually have very simple cell computations and hence short microprograms.

- Word size: Individually programmed processors are subject to a tradeoff in word size: small word sizes lead to an imbalance between the hardware devoted to control and that devoted to data paths, and large words lead to large chips with large pinout and low yield.

Based on the considerations discussed above, the PSC, *a single-chip programmable* systolic processor, has been designed and fabricated in nMOS. Since this processor is an experimental prototype, rather than a production version, the decision was taken to keep the design simple and general, albeit at the cost of pincount and ultimate performance. Nonetheless, as discussed below, the chip as designed already represents a cost-effective means of implementing many systolic algorithms.

Processor structure The processor consists of a collection of communicating functional units, *all of which may operate in parallel.* This collection is made up of a microcode RAM and microsequencer, a register file, an ALU, a multiplier-accumulator (MAC), and three input and three output ports. As schematized in Figure 2-1, data communication among the units takes place on three independent buses; control and status lines are separate. Each bus can be written by one of eight sources and be read by any of ten destinations. This organization supports a significant amount of on-chip parallelism; a multiplication with accumulation, an addition, memory fetches, and interchip I/O can take place concurrently in one instruction cycle. The parallelism-limiting effect of having only three buses, as opposed to eight, is alleviated by the fact that a value on a bus is often used more than once, and by the ability of each of the functional units to hold its inputs over more than one cycle.

Word size In order to keep the chip small and hence keep pinout and yields reasonable, a modest word size of eight bits for arithmetic was chosen. Most data paths in the chip, however, are nine bits wide. The ninth bit can be used to tag data, to store control information, or as the most significant bit of a number modulo 257 for coding applications. In order to support arithmetic on larger numbers, facilities

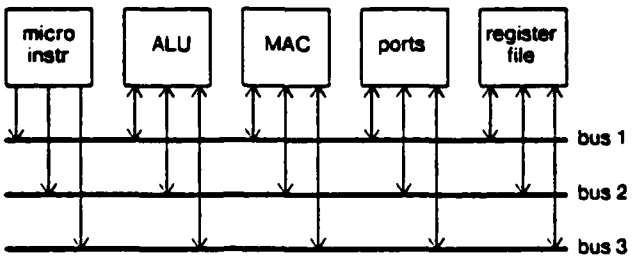


Figure 2-1: Bus structure of the PSC.

are also provided for multiple-precision computation: the multiplier and ALU have provisions for setting a carry in to the low-order bit, and the ALU can cycle its carry out into its carry in for the next operation.

I/O ports Each of three input and three output ports contains nine bits, eight of which are data, with the ninth available for data or control. The ninth bit of each input port is available as a condition code for microprogram branching. This is often used as a tag bit for variable-length data or as a systolic control bit for loading and unloading of stored values. The ninth bit of each output port may be set as a literal, as the most significant bit of a nine-bit bus value, or as the most recent value of an incoming ninth bit.

In order to keep the clock period short, interchip communication is overlapped with instruction execution. Thus a value which is computed in a given cycle can be used on the same chip in the next cycle, but not until the cycle after that on a neighboring chip. The effect on system performance is to increase latency through the array (in clock cycles, though not necessarily in real time), but to reduce the time needed for each pipeline stage.

Control part The microprogram memory consists of 64 60-bit words of dynamic (for circuit density) RAM. The 60 bits in a horizontal microinstruction are divided as follows:

- 9 bits for bus source addresses.
- 20 bits for control of functional unit input registers.
- 9 bits for control of off-chip systolic control bits.
- 10 bits for control of ALU, multiplier, and register file.

- 3 bits to control program branching.
- 9 bits to provide a literal branch address, literal data to the bus, or condition code selection for branching.

The microsequencer is able to fetch instructions in sequence, branch to a literal address, branch to one of four locations depending on any two of twelve condition code bits, and push and pop addresses to and from a subroutine stack.

Arithmetic The ALU performs "standard" arithmetic and logical operations: addition, subtraction, logical AND, etc.. It also allows its carry in bit to be set, either as part of its opcode or as its previous carry out, as an aid to multiple-precision computation. A typical complement of condition code outputs is supplied to the microsequencer. Arithmetic is performed in twos complement notation, except that an "unsigned subtraction" operation is available for eight-bit character comparisons and normalization of numbers modulo 257.

The multiplier-accumulator multiplies two eight-bit numbers and adds them to a 16-bit accumulator, a third eight bit input, or zero to produce a sixteen bit output. The numbers used may be signed (in twos complement) or unsigned, independently.

Register file The register file consists of 64 nine-bit words of dynamic RAM. One word may be read or written in each cycle. In order to remove the RAM's delay from the chip's critical path, the register file is pipelined one cycle behind the rest of the chip; thus an address must be supplied one cycle before the addressed value is needed. This does not appear to pose a performance penalty, since register file accesses seem in general to be uniform and predictable.

Microcode loading All microcode is loaded through a shift register, which can also be used for functional testing of the chip.

In summary, then, the design of the PSC responds to the issues discussed in the previous section as follows:

- Locality and flexibility of control: The microprogrammed control described provides a very high degree of flexibility.
- Chip count: The processor is designed and implemented as a single chip.
- Primitive operations: The processor has features which enable it to per-

form multiplication, other arithmetic operations (including multiple precision), and branching and subroutine control very efficiently.

- Intercell communication: The three sets of I/O ports provide high-bandwidth data flow between neighboring cells, and efficiently support the passing of systolic control bits.

- Internal parallelism: The ports, ALU, multiplier, program memory and data memory all function concurrently. The parallel bus structure allows a number of separate computations to proceed simultaneously.

- Control structure: The horizontal organization of the microcode allows all of the functional units to be utilized in a single cycle.

- Word size: The eight/nine bit format chosen represents a reasonable compromise in terms of size, yield and utility. As indicated below, eight-bit PSCs can already be very useful. Longer words will probably be used for future PSCs, especially as silicon feature sizes shrink, in order to make them useful to a broader class of applications.

A detailed description of the architecture of the PSC is given in Appendix A.

2.1.2. Applications

The PSC can be used as the basic systolic cell for many systolic systems in many application areas. This section gives a flavor of how it is applied in two target applications, and discusses its cost-effectiveness. These and other applications have been implemented in microcode; this code has been simulated by ISPS [4] simulators and used to evaluate architectural specifications of the chip. We estimate that a commercial nMOS implementation of the PSC could easily operate with a cycle time of 200 ns; the timing estimates given below use this figure.

Digital filtering Many digital signal and image processing applications require high-speed filtering capabilities. Mathematically, a filtering problem with $h + k$ taps is defined as follows:

> **given** the weights $\{w_1, w_2, \ldots, w_h\}$, $\{r_1, r_2, \ldots, r_k\}$,
> the initial values $\{y_0, y_{-1}, \ldots, y_{-k+1}\}$
> and the input data $\{x_1, \ldots, x_n\}$,
> **compute** the output sequence $\{y_1, y_2, \ldots, y_{n+1-h}\}$
> defined by

$$y_i = \sum_{j=1}^{h} w_j x_{i+j-1} + \sum_{j=1}^{k} r_j y_{i-j}$$

If the $\{r_i\}$ are all zero, then the problem is called a finite impulse response (FIR) filter; otherwise, it is an infinite impulse response (IIR) filter. Both types of filtering can be performed by systolic arrays [24, 26]. Figure 2-2 shows a systolic FIR filtering array for $h=3$.

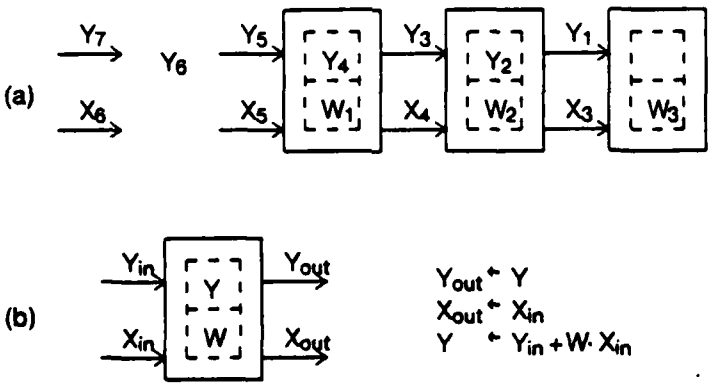


Figure 2-2: (a) Systolic FIR filtering array and (b) its cell definition.

Based on this scheme, digital filters (FIR or IIR) with eight-bit data and weights and m taps can be computed with a linear systolic array composed of m PSCs, taking one sample each 200 ns, for a filtering rate of 5MHz. Thus with 40 PSCs, a 40 tap filter can be computed at a rate of 400 million operations per second (MOPS), counting each inner product step (multiply, add) as two operations.

The PSC program implementing this systolic algorithm takes only one instruction to implement the operations depicted in Figure 2-2(b). After an initialization phase in which the weights are loaded, the inner loop of the algorithm is performed by the PSC microinstruction coded as follows:

```
Bus1=Sda, Bus2=Sdb, Bus3=Lo,
SdaOut=Val3, SdbOut=Val2,
MacX=Hold, MacY=Val2, MacZ=Val1, MacOp=AddZ,
Jump=OnCC0, CC0=Sca, ScaOut=Pass.
```

The lines of this microinstruction have the following effects, all in a single cycle, as sketched in Figure 2-3:

1. Bus 1 carries Y_{in}, read from systolic data port A, bus 2 carries X_{in}, read from port B, and bus 3 carries Y_{out} from the previous operation, available as the output of the MAC.

2. Output port A receives Y_{out} from the previous operation (the value on bus 3), and port B receives $X_{out} = X_{in}$.

3. The MAC holds the cell's weight in its x register, sets its y register to X_{in} (the value on bus 2), and sets its z register to Y_{in} (the value on bus 1). It then computes $x \cdot y + z$, or $W \cdot X_{in} + Y_{in}$. This value will be sent to output port A during the next cycle.

4. The instruction loops in place until systolic control signal A arrives, meaning it is time to reinitialize. Control then passes to the next instruction, and the control bit is sent on to the neighboring cell.

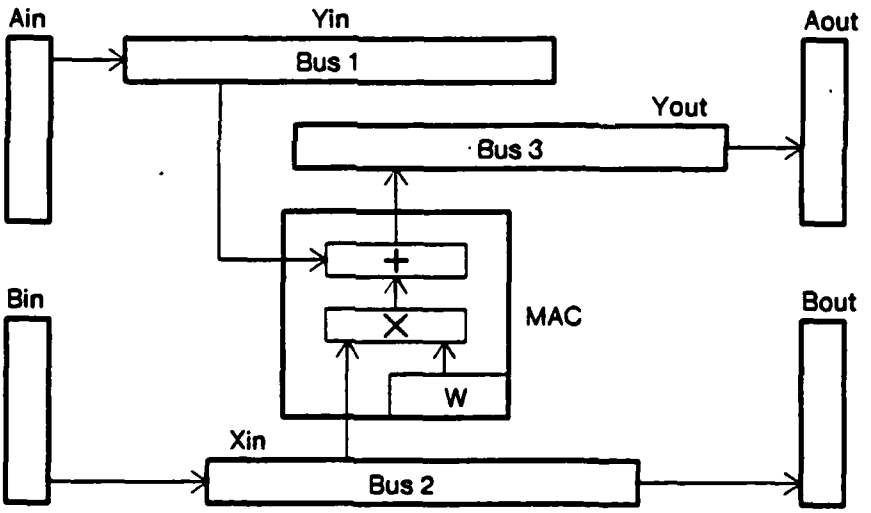


Figure 2-3: PSC implementation of FIR filter.

For applications requiring more accuracy, a filter with 16-bit data and eight-bit coefficients and m taps can be computed with m PSCs, taking one sample each 1.2 μs (six instructions). Thus with 40 PSCs, a 40 tap filter can be computed at a rate of 67 MOPS, counting each inner product step as two operations. This is equivalent to 200 MOPS for eight-bit arithmetic.

Error-correcting codes Among various error-correcting codes, Reed-Solomon codes are most widely used for deep space communications, where burst errors occur frequently [36, 42]. One popular Reed-Solomon code is a scheme in which a codeword consists of 224 message symbols followed by 32 parity symbols. This code can correct up to 16 symbol errors per codeword through a decoding process.

Each symbol is defined over a finite field of 257 elements, denoted by $GF(257)$, and thus is encoded with 9 bits. Any number less than 256 is represented as usual by 8 bits, and the number 256 ($\equiv -1$) by 9 bits as 100000000_2. We call this representation "normalized form". Arithmetic is performed on numbers in normalized form, and gives a result in normalized form. The basic pattern is to test if one of the operands is equal to 256—in which case a special treatment is applied—then operate on 8 bits and normalize the result.

Encoding a message, i.e., obtaining parity symbols from the given message symbols, is equivalent to a polynomial division. It can therefore be carried out with a systolic division array described by Kung [25]. Since in this case the divisor—the *generator polynomial* in the terminology of error-correcting codes—is monic, no cell in the systolic array needs to perform a numerical division. Each cell is basically the same as used in the systolic filtering array discussed above, except that integer arithmetic modulo 257 is used.

Decoding, which is much more complex than encoding, is usually done in four steps. We will not describe the steps in detail here; the reader is referred to the texts on error-correcting codes cited earlier. In the following we simply point out that each step corresponds to some polynomial computation over the finite field $GF(257)$ that can be effectively carried out by systolic arrays.

- *Syndrome computation* is to compute the syndrome polynomial $S(x)$ of degree 31. The problem is equivalent to that of evaluating a polynomial of degree 255 at 32 points. Using Horner's rule, this can be done by a systolic array where each cell holds one of the 32 points and accumulates results as the coefficients of the polynomial flow through the array [24].

- *Solution of the key equation* is to find two polynomials, the error locator polynomial $\omega(x)$ and error evaluator polynomial $\sigma(x)$, with $\deg \omega < 16$ and $\deg \sigma \leq 16$, such that the key equation,

$$\omega(x) \equiv \sigma(x) S(x) [\bmod x^{32}],$$

is satisfied. This problem can be solved by a systolic array for computing extended greatest common divisors [8].

- *Error location* is to find the roots of $\sigma(x)=0$, which will identify the locations of errors in the received message. Finding the roots is most efficiently done by simply evaluating $\sigma(x)$ at every point in $GF(257)$, since the size of the field is small. Again as in the syndrome computation, based on Horner's rule we can use a systolic array for carrying out the polynomial evaluation. Now since the degree of the polynomial is small and the number of the points where the polynomial is to be evaluated is large, we use a "dual" systolic design where each cell holds one coefficient of the polynomial and the points flow along the array.

- *Error evaluation* is to evaluate the amplitude of each error found in the previous step, by evaluating $\omega(x)/\sigma'(x)$ at the roots of $\sigma(x)=0$. This again calls for polynomial evaluation.

Figure 2-4 illustrates that all the steps mentioned above can be implemented with appropriate systolic arrays made up of the same PSCs, with different microcode for each array. We estimate that by using a linear array of 112 PSCs, Reed-Solomon decoding can be performed with a throughput of 8 million bits per second. Encoding is much easier; it requires only about 16 PSCs to achieve the same throughput. At the time the PSC was designed, the fastest existing Reed-Solomon decoder with similar mathematical characteristics used about 500 chips but achieved a throughput of no more than 1 million bits per second.

2.1.3. Implementation

The PSC architecture was implemented in 4μ nMOS [11] by a team of chip designers [16], some part-time and some full-time. It was estimated that a commercial nMOS implementation could run with a 200 ns cycle time, and that the initial implementation would have a cycle time of about 350 ns. Working chips were obtained on the second fabrication run. Due to shifts in the parameters of available fabrication and noise sensitivity in some parts of the circuit design, along with limited timing precision of the testing equipment used, actual cycle times averaged about 1.5 μs, with the fastest observed cycle time being about 700 ns. These cycle times are consistent with earlier estimates, taking into account the parameter shifts,

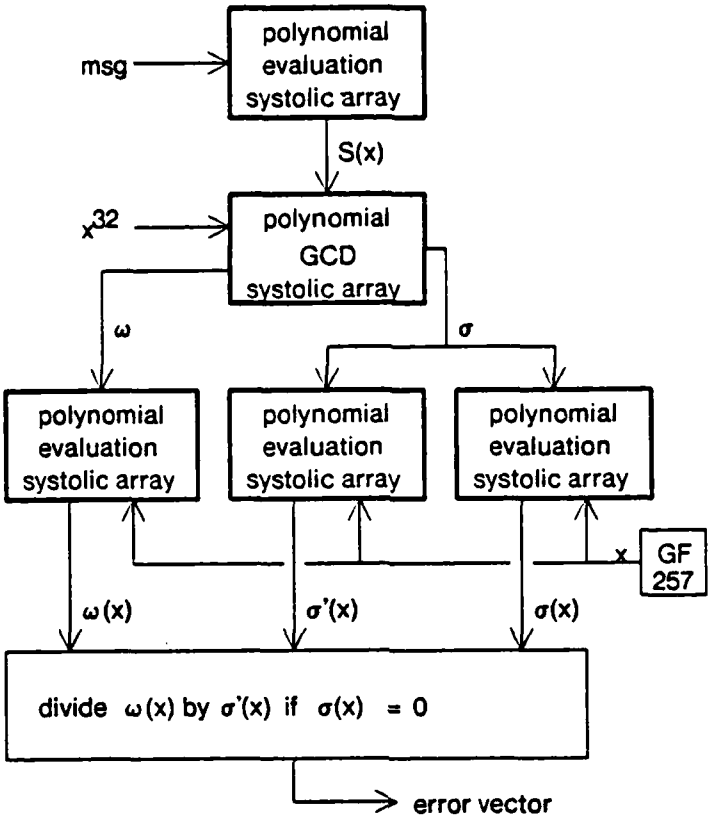


Figure 2-4: Systolic decoder for Reed-Solomon code.

accommodations to noise sensitivity, and tester speed. Using these chips, a system which performs filtering of video images has been constructed [17].

2.1.4. Implementation alternatives

The PSC represents only one possible means of implementation of systolic algorithms. Alternatives include board-level implementation from existing parts, full custom single-purpose LSI implementation, and implementation using existing microprocessors.

Board-level and full custom LSI implementations may be preferable to the PSC approach where performance requirements are very stringent. The disadvantages of these approaches are mainly in design cost (especially for single-purpose custom LSI) and, for board-level implementations, costs of packaging, power dissipation, and physical size.

The alternative approach closest in spirit to the use of PSCs is the use of existing microprocessors. Most microprocessors would perform very poorly in this context—in many cases, the PSC works an order of magnitude faster. The reasons for this failure reside in the differences between the PSC and conventional μPs: I/O facilities, on-chip program memory, internal parallelism, parallel multiplier and systolic control bits. No commercially available μP that we are aware of combines all of the above properties. Instead, many microprocessors have only one or two of those features.

Commercially available microprocessors can be divided into four groups:

- *"Standard" microprocessors* with word sizes of eight bits (Z80, 6800, 8080 families) or larger (68000, 16000 families), have no internal program or scratch pad RAM (usually a few registers), no hardware multiplier, and at most two ports: an address port and a data port. In order to be used in systolic arrays, each processor would need external memory chips, I/O devices and a few components for inter-chip communication. Such devices could be built, but at high cost in chip count, power dissipation, and space. Also, conventional microprocessors provide large instruction sets and complex addressing schemes, but cannot be programmed at the microcode level; *as a result, even simple operations take several cycles.* Finally, I/O is a serious bottleneck since the same ports are used for fetching operands and instructions and for inter-chip communication.

- *Signal processing microprocessors* (NEC 7720, TI TMS320, Hitachi HSP) usually have an on-chip program memory in ROM, a small data memory, and (for the most recent ones) a parallel multiplier. They also show some amount of internal parallelism since they can perform a multiplication-accumulation and increment an address register at the same time. However, their structure is often too specialized to use them in applications other than signal processing, and the I/O problem is also present; none have the off-chip bandwidth necessary for systolic array communication.

- *Single-chip micro-computers* (8020, 8749) are, in some respects, superficially similar to the PSC. They have on-chip ROM program memory, a small RAM for data, and up to three bidirectional ports (used serially, however, not simultaneously). They can be used with few external components, as opposed to microprocessors that need an external memory and I/O devices in order to function. However, they are serial machines with simple instruction sets and without multipliers, and suffer from the same I/O bottleneck as traditional microprocessors.

- *Bit-slice microprocessors* form a last family. A PSC equivalent could be

designed with bit-slice processors and many external parts, and this could have been an alternative to the implementation of a chip. However, the result would be uneconomical in most applications because of the large chip count and accompanying size, fabrication, and power costs.

As an example of the effects of serial execution and data access, consider the filtering example of Figure 2-2. A microprocessor equipped with a multiplier and the ability to perform an I/O operation to either of its neighbors in a single cycle would need at least 10 instruction cycles to perform this inner loop (4 for I/O and 6 for branching, arithmetic, and data movement). Counting the overhead of instruction fetching and decoding, it is extremely unlikely that such a processor, implemented in similar technology, would have a smaller instruction cycle time than the PSC; hence the PSC's organization allows it to operate at least ten times faster in this case. For Reed-Solomon encoding, a standard μP with a multiplier added would execute at least four times as many instructions as the PSC. Most systolic algorithms seem to produce instruction ratios in the range between 4:1 and 10:1.

It is evident from these considerations that current microprocessor architectures are not well-suited to the construction of systolic arrays. The key points, again, that favor the PSC structure are:

- The large number of ports (including systolic control) used in parallel. Note that each one of these ports offers more data bandwidth than the usual microprocessor port, which for each data transfer will usually transfer an instruction address, an instruction, and a data address.

- Internal program and data memory: I/O bandwidth is fully available for transfer of data between chips. In our examples, the ratio (number of input+output)/(number of instructions) is high, often larger than 1. Also, typical programs are short and can fit into a small control memory.

- Horizontal microprogramming: large microinstructions provide effective internal parallelism. This is possible only because of the rich interconnection pattern (3 internal global buses).

- Systolic control: this is indispensable for systolic algorithms, and is costly in instruction cycles and I/O bandwidth for conventional μPs.

- Multiplier: this is critical for high performance in numerical applications.

2.1.5. Limitations and improvements

For simple computations, the PSC's arithmetic, internal communication and external communication capacities are fairly well balanced. For more complex algorithms, the most common limiting factor in program performance is the number of buses. Adding more buses would be quite expensive in area needed for routing and for code storage and distribution; for a general-purpose part, this expense would probably not be justified. Other possibilities would be to use some specialized buses, to allow a single bus to be broken into independent parts, or to multiplex the use of the buses within a machine cycle. One other option, which would be useful in the frequent case where a value needs to be delayed as it enters or leaves a cell, would be to put small FIFOs on the chip's I/O ports.

Another limitation is in the size and bandwidth of the register file. Only one word can be read or written in a cycle, making the storage and retrieval of intermediate results time-consuming compared to computation. Furthermore, the small size of the file (64 words for the current implementation) limits the ability of the PSC to take advantage of memory space to cut down on I/O traffic (e. g., by storing an entire column of a matrix in a cell). Possible improvements include the use of multiported registers or off-chip memory.

One way to reduce the cost of a PSC-based system would be to reduce the chip's complement of I/O pins, 54 of which are dedicated to the data ports. The PSC's use of three input and three output ports is due mainly to simplicity considerations: almost all systolic algorithms' data flows can be implemented in a straightforward way with a minimum of control. Since all six ports are needed simultaneously only for rather simple algorithms where communication dominates computation by a large factor, it may be possible to reduce the pincount of the chip without greatly reducing its overall performance. This could be achieved by using bidirectional ports (perhaps four), at a modest cost in control complexity. This would reduce the total I/O bandwidth of the chip by 33%, but increase its interconnection flexibility.

Another area where the cost of the PSC might be reduced is in microcode space. Again for reasons of simplicity and flexibility, no attempt was made to squeeze the microinstruction size by limiting the number and kind of operations the PSC's paral-

lel functional units could perform. While this is a useful property for an experimental system, it would be advantageous for production chips to sacrifice some flexibility for higher yield (due to smaller size) or more words of data or instruction memory.

2.2. Alternative memory structures

This section deals with two different approaches to the structuring of memory and processing in systolic implementations. Section 2.2.1 describes the known technique of using local memory within cells, and clarifies its application. It then considers questions of modularity and processor/memory tradeoffs, improving on some previous designs and developing criteria for deciding how to design chips for a given set of applications.

Section 2.2.2 describes a structure which can be used to implement systolic algorithms with data flow separated from computation. This separation allows the creation of modular, general-purpose systolic hardware; new processing units can be created using off-the-shelf parts. The memory structure can be built with local communication only or using technologies which are inherently serial, making it usable for very large sets of data.

2.2.1. Local memories and modularity

Local memories may be used within systolic processors for two purposes: reduction of I/O bandwidth and multiplexing of processing units to solve large problems on smaller hardware. I/O bandwidth can usually be reduced without reducing performance when data are used repeatedly in a regular fashion. Multiplexing can be applied to any systolic algorithm, and results in a performance, processor count, and I/O bandwidth decrease which is proportional to the degree of multiplexing.

Processor multiplexing Throughput and hardware may be linearly traded off in a systolic array design by "compressing" the array along one or more dimensions. For example, a linear array can use half as many cells, each with enough storage to simulate two of the original cells using twice as much time. Given that c original cells are replaced by one, in effect slowing each original cell by a factor of c, the new

design has its throughput and I/O bandwidth also reduced by a factor of c. An example of a programmable machine using this well-known scheme is the CMU Warp processor [27].

One interesting effect arises when compressions of two-dimensional (or higher) arrays are considered. We will consider only rectilinear arrays; note that hexagonal arrays are rectilinear arrays with diagonal connections. The array can be compressed uniformly in each of its dimensions; total I/O bandwidth scales with processor count, but the I/O bandwidth of each cell depends on the direction of compression. As sketched in Figure 2-5, data flowing in the direction of compression are slowed by a factor of c. Data flowing across the direction of compression are slowed by the same factor, but each cell now supports c data streams, so the cell's bandwidth with respect to that set of data remains the same.

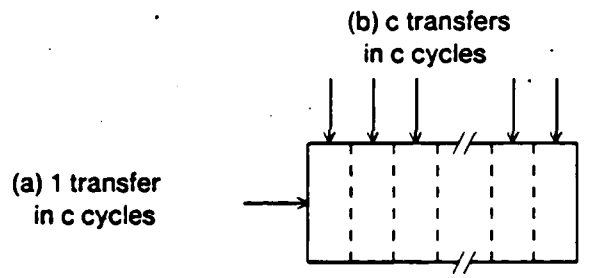


Figure 2-5: (a) Data in direction of compression and (b) data across compression.

One result of this effect is that rectangular arrays are best compressed to smaller arrays of the same aspect ratio, rather than to linear arrays. Consider the compression of an $n \times n$ array, each cell having unit bandwidth in each dimension, by a factor of n. If the array is compacted by $\sqrt{n}$ in each dimension, each cell has I/O bandwidth $1/\sqrt{n}$ in each dimension. If the array is compacted to a linear array, each cell has unit bandwidth along the length of the array, and bandwidth $1/n$ transversely. In the square case, cells can time-multiplex their inputs to allow cheaper implementations; in the linear case, this is not possible.

When the problem at hand is the decomposition of a large array onto chips, the "square" approach is still preferable. Figure 2-6 shows two such decompositions

with k processors per chip, one where compression is performed in two dimensions and one where it is performed in one dimension. A square chip has bandwidth proportional to $\sqrt{k}/\sqrt{c}$, while the linear chip has bandwidth proportional to $1 + k/c$. Assuming a fixed value for c, the square chip is increasingly preferable with increasing circuit density (hence increasing k); it also requires only local communication on-chip, while the linear array will need long on-chip wires for its transverse data transfers.

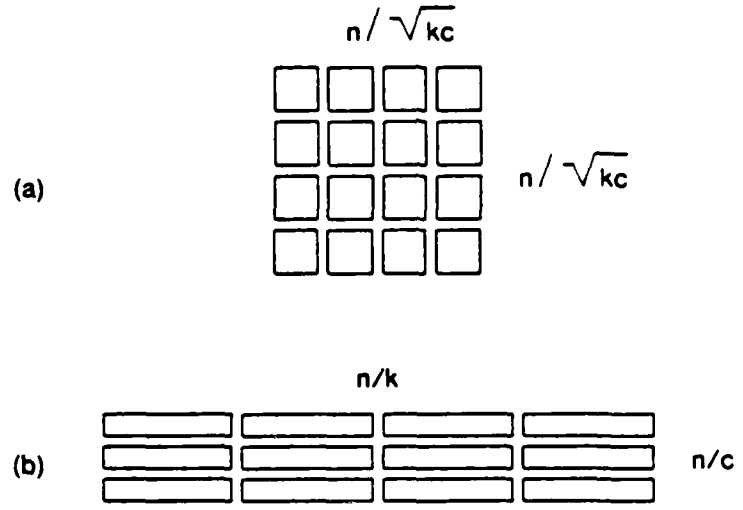


Figure 2-6: (a) Square decomposition and (b) linear decomposition.

Bandwidth reduction Local memory may also be applied in systolic arrays to reduce I/O bandwidth without any reduction in throughput. This is possible where a particular cell uses the same data repeatedly, as when a variant of the matrix-vector multiplication algorithm of Kung and Leiserson [29] is used to apply the same transformation to a number of vectors. In this case, the original algorithm consists of a linear array with one matrix column entering each cell from the broad side of the array, skewed so that each element meets the appropriate element of the vector as it is sequenced through the length of the array. The modification is simply to store each column of the matrix in the corresponding cell, so that only unit I/O bandwidth at the ends of the array is required.

One criticism that can be leveled at this approach is that any straightforward

implementation is not modular; that is, a matrix with more rows than a cell has words of memory cannot be handled in a simple way. Ramakrishnan and Varman [43] point out that this problem can be solved by using a linear array of $\theta(n^2)$ (actually $3/2n^2 + O(n)$) cells, with no additional memory. The exact problem that they address is matrix multiplication, which can be thought of as multiplication of n vectors by a single matrix. While this approach is asymptotically hardware-optimal in the sense that an array must be capable of storing n^2 values, it is impractical in that the arithmetic unit of each inner-product cell sits idle for approximately $n-1$ out of n cycles.

As a practical matter, we should note that the modular scheme just mentioned is unlikely, at least when applied in a straightforward way, to be preferable to an internal-memory implementation. If we note that a combinational multiplier takes up circuit area greater than that needed to store 100 words of memory (and perhaps even 1000), we see that a 100×100 memory-based multiplication array will take no more than 200 multiplier-equivalents, while a "modular" array with the same performance will take about 15000 — 75 times as much. Nonetheless, two methods may be used to improve upon the modular scheme. One is based on considering the bit complexity of the arithmetic operations used, and the other is based on reintroducing memory.

For the first method, suppose that the algorithm operates on b-bit integers. If parallel multipliers are used, the algorithm actually uses $O(n^2b^2)$ hardware; if serial-parallel multipliers are used, the algorithm takes $O(n^2b)$ steps. In either case, the optimality result based on fixed word length no longer holds. In addition, when any fixed-precision number representation is used, no bounded-I/O scheme can have constant throughput, as the magnitudes of the results grow as $O(\log n)$. Henceforth, we will assume that $b=\Omega(\log n)$ is large enough to accommodate any number that can result in the matrix multiplication.

The algorithm can be restored to optimality, in the case where $n > b$, by using b-bit adders instead of parallel multipliers. Ramakrishnan and Varman's algorithm can be adapted directly by distributing the accumulation of a product over b cells, but it is easiest to consider a variation where one matrix sits motionless in the array and the

other passes by. (This is not detrimental even for the case where a different pair of arrays is used each time, since a new array can be inserted in double-buffered fashion during the previous computation.) To compute $A \times B$, the multiplier matrix A is stored in row-major order in the array, and the multiplicand matrix B passes through in column-major order (see Figure 2-7). Each cell, upon receiving an element of the multiplicand which needs to be multiplied by its stored value, spends b cycles multiplying while irrelevant multiplicand elements pass by. N cycles after it received its first multiplicand value, it receives another and begins again to multiply. Result values are accumulated as they pass through the cells holding elements of one row of A. This method uses $O(n^2 b)$ hardware and $O(n^2)$ steps, which are optimal for an array of I/O bandwidth b. It has the additional benefit of reducing cycle time from that required for a parallel multiplication to that required for an addition. This approach can also be applied to other number formats (e. g. floating point), by similarly breaking the inner product operation into smaller steps.

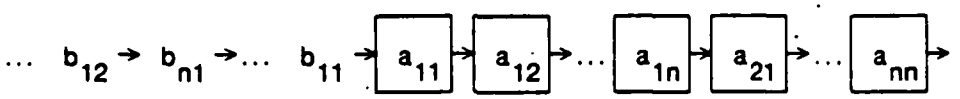


Figure 2-7: Matrix multiplication algorithm.

The second method for improving on the modular algorithm is to use memory. In the first method, we increased processor utilization by a factor of b by making each processor less powerful (and less expensive). Using memory, we can perform the compression trick of the previous section, but instead of reducing throughput we reduce idle cycles. Given that available parts have m words of memory per processor, $n \times n$ matrix multiplications where $n \geq m$ will require $\lceil p_0/m \rceil$ processors, where p_0 is the number of processors used in the original algorithm. The result is a large constant factor decrease in hardware over the original algorithm, with the same performance.

2.2.2. Decoupled data flow

The usual view of systolic arrays is modular in the sense that an array can be enlarged by adding more identical cells. On the other hand, such arrays are not modular in the sense that computing power and data storage are inseparable, unlike processor and memory boards in a conventional system. What this means is that the effort put into designing a system cannot ordinarily be partially or fully recouped by reusing either its computational part or its data flow part for another purpose.

One way to achieve modularity in the second sense would be to isolate cell processing in a separate processing unit, and to arrange for it to sequence through a set of cell states. The mode of operation of such a system would be to compute new values for each cell, and then to perform data movements corresponding to those in the abstract array being modeled. Assuming for the moment that data access and movement is easy, this organization has several attractive features. First, it allows changes in the type of processing performed in a cell without reimplementing the data access portion of the machine. Second, since there are no dependencies between the individual cell computations in a given systolic step, the processing unit may be highly parallel and pipelined. Third, it makes it easy to use different implementation technologies for processing and for memory, allowing for fast processing and small storage size.

A systolic data unit Missing from the description above is a mechanism for data access and movement. For linear arrays, this is straightforward; one shift register (perhaps bidirectional), easily implemented with RAM, can be used for each data stream. Implementation with a global RAM becomes more difficult for two-dimensional arrays. In this case, data access requires potentially complex address calculations, and data movement requires large memory updates or clever pointer schemes, especially in the case of "diagonal" flow.

An alternative to a global RAM implementation is the use of a shift register with a major loop/minor loop structure, as found in magnetic bubble memories. For an $n \times n$ array, the structure consists of n loops of n cells, all connected by a major loop of n cells. One such structure is used for each data stream in the algorithm. Depending on the direction of data flow (vertical, horizontal, or diagonal), data are

arranged in such a way that shifting along minor loops (with appropriate insertions/deletions for input and output) models data movement. Data access is achieved by loading data from minor loops to the major loop at appropriate times. These operations can be supervised by a programmed controller at each minor loop, or they can be signalled by control tokens that travel with the data. Figure 2-8 illustrates horizontal, vertical, and diagonal data flows for the 4×4 case.

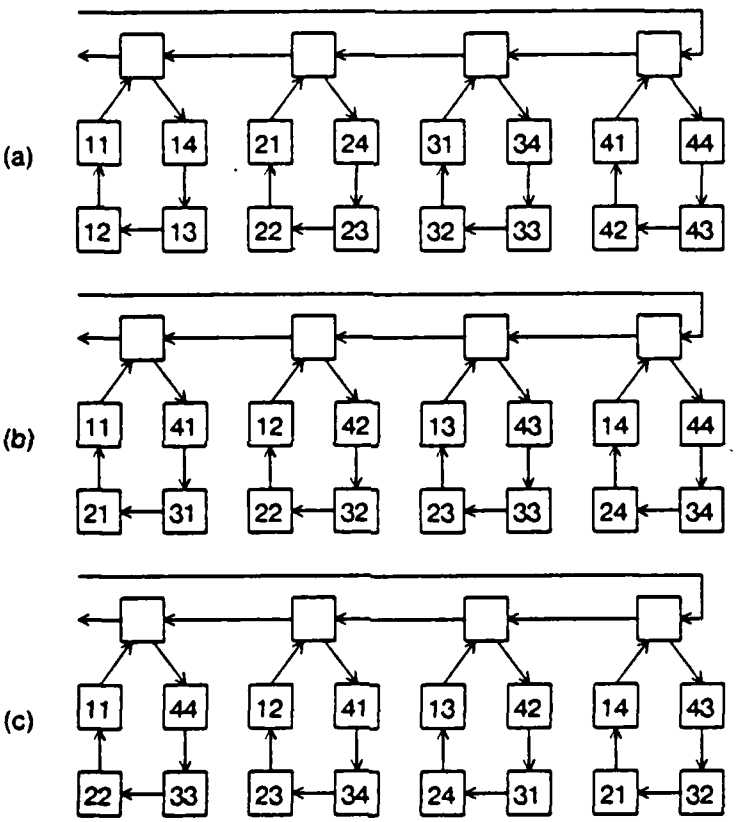


Figure 2-8: (a) Horizontal, (b) vertical and (c) diagonal data flows.

Horizontal data streams are stored with one row in each minor loop. To access the data during computation, the minor loops are threaded together so that access is performed in row-major order. Vertical data streams are stored with one column in each minor loop, and row-major access is performed by loading data from the minor loops to the major loop and shifting them out. Diagonal data streams are stored in a similar fashion; diagonals other than the main diagonal share minor loops in such a

way that each minor loop has one element from each row and each column, ordered in such a way that they can be fed out to the major loop in row-major order.

Output from the emulated systolic array is done by taking the appropriate values as they go by, and advancing their successors in the shift sequence to the leading edge position; input is done by inserting new values at the trailing edge position as the shift register is reloaded. The exact way in which reloading happens depends on the latency of cell processing. If this latency matches the cycle time of the data unit (which is unlikely), reloading matches unloading in a straightforward way. If the latency is longer than the data unit cycle time, slack can be provided by allowing the minor loops to "shrink", rather than loading a new value every time an old value is shifted out. This feature is likely to exist in any reasonable implementation, since it is needed to provide for arrays of varying size.

The most straightforward implementation of this structure, using shift register circuitry, is probably not the most economical. A less expensive but somewhat slower approach would be to use a linear array of n memory processors, each with storage for n values. Address generation for each processor is reduced to the one-dimensional case, and adaptation to smaller arrays is easy. Furthermore, each memory is quite a bit smaller than in the monolithic case, so access may be faster. Note that all communication among the memory processors is local.

In summary, this structure provides an efficient and general mechanism for the maintenance and access of systolic data streams. This allows the construction of systolic machines whose computational mechanism is separate from data access. This in turn allows separate optimizations; the throughput of a system may be increased by adding more processing power (assuming available data unit bandwidth), or a system may be enabled to handle larger problems by adding more memory, without changing throughput or I/O bandwidth. The shift register structure may be implemented in a number of ways, either by actual shift registers (for ultimate performance) or by RAM implementations of shift registers.

2.3. Conclusions

The PSC The architecture of the PSC is very well-suited to the implementation of a wide variety of systolic algorithms with simple inner loops. The 64 instruction words available accommodate most straightforward computations, and suffice even for such complex cell computations as those found in the systolic Reed-Solomon decoder. The processor's control structure imposes very little overhead relative to the arithmetic heart of the computation; instruction fetch occurs in parallel with execution. For algorithms with complex cell calculations, greater speed can be gained by using multiple PSCs for each cell. Concurrency and parallel I/O result in much smaller instruction counts than for conventional microprocessors. Finally, the incorporation of control and communication circuitry onto the same chip as the arithmetic units achieves a large savings in chip count over systems built with standard parts. A PSC equivalent built with LSI arithmetic, memory and control and with TTL latches and multiplexers would require on the order of one hundred chips, and a similarly constructed single-purpose cell for even a slightly complicated algorithm would require a dozen chips.

In a broader context, a PSC-like architecture may be viewed as the logical next step in the evolution of arithmetic components. When construed as an intelligent multiplier (that is, a multiplier with ancillary arithmetic, buffering and interconnection, and powerful control facilities) rather than as a simple-minded processor, it fits well into many types of systems as a computational building-block. Just as the introduction of the multiplier accumulator [46] made it possible to implement many circuits with fewer parts than previously, so a PSC-like component could reduce chip counts and add flexibility to many types of machines.

Memory issues The one-to-one binding of processors and memory generally used by the PSC (with the exception of some uses of its register file) is only one way of organizing systolic computations. As Section 2.2 shows, structures in which many data are associated with a processor or in which processing and storage are completely separated may prove advantageous given certain application properties.

3 Synchronization

In order for the elements of a processor array to communicate among themselves, some provision must be made for synchronization of data transfer. The conceptually simplest means of synchronization is the use of a global clock. Unfortunately, large clocked systems can be difficult to implement because of clock skews and delays, which can be especially acute in VLSI systems as feature sizes shrink. For the near term, good engineering and technology improvements can be expected to maintain the feasibility of clocking in such systems; however, clock distribution problems crop up in any technology as systems grow. An alternative means of enforcing synchronization is the use of self-timed, asynchronous schemes, at the cost of increased design complexity and hardware cost. On the premise that different circumstances call for different synchronization methods, this chapter provides a spectrum of synchronization models; based on the assumptions made for each model, theoretical lower bounds on clock skew are derived, and appropriate or best-possible synchronization schemes for large processor arrays are proposed.

One set of models is based on assumptions that allow the use of a pipelined clocking scheme, where more than one clock event is propagated at a time. In this case, it is shown that even assuming that physical variations along clock lines can produce skews between wires proportional to their lengths, any one-dimensional processor array can be correctly synchronized by a global pipelined clock while enjoying desirable properties such as modularity, expandability and robustness. This result cannot be extended to two-dimensional arrays, however—we will see that under this assumption, it is impossible to run a clock such that the maximum clock skew between two communicating cells will be bounded by a constant as the system

grows. For such cases or where pipelined clocking is unworkable, a synchronization scheme incorporating both clocked and "asynchronous" elements is proposed. Finally, the practical implications of these results are discussed.

3.1. Introduction

In describing a processor array algorithm, it is often convenient to picture the processors as running in lock step. This *synchronized* view, for example, often makes the definition of the structure and its correctness relatively easy to follow: computations proceed in discrete steps which may be cleanly characterized. Perhaps the simplest means of synchronizing an ensemble of cells is the use of broadcast clocks. A clocked system in general consists of a collection of functional units whose communication is synchronized by external clock signals. A variety of clocking implementations are possible; the essential point is that by referring to the global time standard represented by the clock, communicating cells can agree on when a cell's outputs should be held constant and when a cell should be sensitive to its input wires. When different cells receive clock signals by different paths, they may not receive clocking events at the same time, potentially causing synchronization failure. These synchronization errors due to clock skews can be avoided by lowering clock rates and/or adding delay to circuits, thereby slowing the computation. The usual clocking schemes are also limited in performance by the time needed to drive clock lines, which will grow as circuit feature size shrinks relative to total circuit size. Therefore, unless operating at possibly unacceptable speeds, very large systems controlled by global clocks can be difficult to implement because of the inevitable problem of clock skews and delays.

As a practical aside, we should note that at current LSI circuit densities, clock distribution is still a solvable problem. Two somewhat pessimistic studies of which we are aware [19, 31] do not take into account either the tricks that a circuit designer can use to reduce the RC constant of his clock tree, or the promise of multiple-layer metallization and low-resistance silicides. Given these factors, it seems that the usual clocking schemes should remain feasible for on-chip synchronization in the near term. Moreover, for some specific structures, such as one-dimensional arrays, clocking can be effectively used even in the presence of large signal propagation delays (see Section 3.5.1).

An alternative approach to clocking is self-timing [44], in which cells synchronize their communication locally with some variety of "handshaking" protocols. It is easy to convince oneself that any synchronized parallel system where cells operate in lock step can be converted into a corresponding asynchronous system of this type that computes the same output— the asynchronous system is obtained by simply letting each cell start computing as soon as its inputs become available from other cells. The self-timed, asynchronous scheme can be costly in terms of extra hardware and delay in each cell, but it has the advantage that the time required for a communication event between two cells is independent of the size of the entire processor array. A serious disadvantage of fully self-timed systems is that, given current digital design methodology, they can be difficult and expensive to design and validate.

An advantage that self-timed systems often enjoy, in addition to the absence of clock skew problems, is a performance advantage that results from each cell being able to start computing as soon as its inputs are ready and to make its outputs available as soon as it is finished computing. This allows a machine to take advantage of variations in component speed or data-dependent conditions allowing faster computation. This advantage will seldom exist in regular arrays such as systolic systems, however, for two reasons:

- Usually, each cell in a regular array performs the same kind of computation as every other cell; thus there is little opportunity for speed variation.

- In cases where variations do exist, the throughput of computation along a path in an array is limited by the slowest computation on that path. The probability that a worst-case computation will appear on a path with k cells is $1-p^k$, where p is the probability that any given cell will *not* be performing a worst-case computation. This quantity approaches unity as k grows, so large arrays will usually be forced to operate at worst-case speeds.

The result of these considerations is that clocking is generally preferable to self-timing in the synchronization of highly regular arrays. This chapter derives techniques for synchronizing large arrays, using clocking where possible and preserving some of the advantages of clocked schemes where clocking breaks down.

3.2. Basic assumptions

The basic model that we will use for considering synchronization of VLSI processor arrays is as follows:

(A1) Inter-cell data communications in an *ideally synchronized* processor array, in which all processors operate in lock step, are defined by a directed graph COMM, which is laid out in the plane. Each node of COMM, also called a *cell*, represents a cell of the array, and each directed edge of COMM, called a *communication edge*, represents a wire capable of sending a data item from the source cell to the target cell in every cycle of the system. Any two cells connecting by a communication edge are called *communicating cells*.

(A2) A cell occupies unit area.

(A3) A communication edge has unit width.

We now add assumptions which provide the basis for clocked implementations of ideally synchronized arrays.

(A4) A clock for a clocked processor array is distributed by a rooted binary tree CLK, which is also laid out in the plane. A cell of COMM can be clocked if the cell is also a node of CLK.

(A5) A clocked system may be driven with clock period $\sigma + \delta + \tau$, where σ is the maximum clock skew between any two communicating cells, δ is the maximum time for a cell's outputs to be computed and propagated to a communicating cell, and τ is the time to distribute a clocking event on CLK.

This assumption is an abstraction of properties common to all clocking schemes. The detailed relationships between these parameters and other more specific parameters such as flip-flop setup and hold times depend on the exact clocking method used. An exact representation of minimum clock period might be something like $\max(\tau, 2\sigma + \delta)$ in a particular case, but such formulae will exhibit the same type of growth with respect to system size as the simple sum used here.

Note that if we adopt the usual convention that the clock tree is brought to an equipotential state before a new clock event is transmitted, eliminating clock skew can lead only to a constant factor increase in performance, since it must always be true that $\sigma \leq \tau$. In particular, speed of light considerations impose the following condition:

(A6) The time τ required to distribute a clocking event on a clock tree CLK in a particular layout is bounded below by $\alpha \cdot P$, where $\alpha > 0$ is a constant and P is the (physical) length of a longest root-to-leaf path in CLK.

Thus, since the clock tree must reach each cell in the array, large arrays which are synchronized by equipotential clocking must have clock periods at least proportional to their layouts' diameters. Note that in the remainder of this chapter, we will relate transmission delays to wire length; delays are caused by other factors, of course, but we choose to treat them together as a "distance" metric.

In the case where an array grows too big for its clock tree to be driven at the desired speeds due to the time needed to bring long wires to an equipotential state, it is possible to take advantage of the propagation delay down a long wire by having several clock cycles in progress along its length. This mode of clocking is often used in large mainframe computers, but not, to our knowledge, on chips. The electrical problems of passing a clean signal on a chip in this fashion are severe, due to analog phenomena such as damping and reflections. We can instead simulate this behavior by replacing long wires with strings of buffers, which will restore signal levels and prevent backward noise propagation. These buffers are spaced a constant distance apart; a good candidate is that distance which will cause wire delays between buffers to be of the same size as a buffer's propagation delay. This allows us to replace assumption (A6) with the following:

(A7) If CLK is a buffered clock tree, the time τ required to distribute a clocking event on a particular unbuffered segment of CLK is the maximum delay through a buffer and its output wire. Thus, τ is a constant independent of the size of the array.

To ensure that successive clock events remain correctly spaced along the clock path, we make the following assumption:

(A8) The time for a signal to travel on a particular path through a buffered clock tree is invariant over time.

The following section describes two clock skew models based on the above assumptions, and Sections 3.4 and 3.5 explore the problem of clocking under these models. Section 3.6 considers the case where assumption (A8) does not hold, and hence condition (A6) holds rather than condition (A7). Section 3.7 briefly discusses the practicality of the models and the results obtained.

3.3. Two models of clock skew

Given a basic model consisting of conditions (A1) through (A5), plus (A7) and (A8), the following sections consider the implications of two models of clock skew. First, in Section 3.4 we consider the case where clock skew between two cells depends on the difference in their physical distances from the root of the clock tree. This *difference model* corresponds reasonably well with the practical situation in high speed systems made of discrete components, where clock trees are often wired so that delay from the root is the same for all cells. Formally, we assume the following:

(A9) The clock skew between two nodes of CLK, with respect to a given layout, is bounded above by $f(d)$, where f is some monotonically increasing function and d is the positive difference between the (physical) lengths of the paths on CLK that connect the two nodes to the root.

This assumption is illustrated in Figure 3-1. The two circles connected by the dashed line have clock skew between them which is no more than $f(d)$, where d is the length of the crosshatched segment. This segment represents the difference between the cells' distances to their nearest common ancestor in the clock tree.

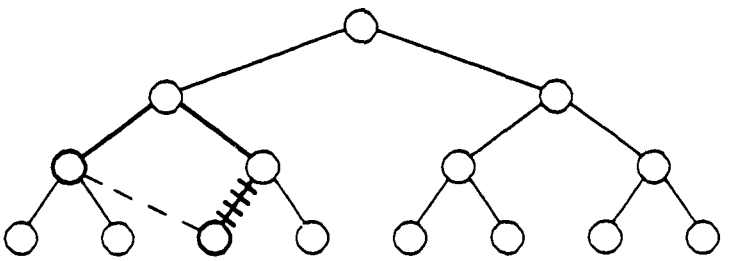


Figure 3-1: Skew in the difference model.

As systems grow, small variations in electrical characteristics along clock lines can build up unpredictably to produce skews even between wires of the same length. In the worst case, two wires can have propagation delays which differ in proportion to the sum of their lengths. Especially since it is not possible to tune the clock network of a system on a single chip, Section 3.5 considers a model in which the skew between two nodes depends on the distance between them along the clock tree. Formally, the *summation model* (so called because the distance between two nodes is the sum of their distances from their nearest common ancestor, while the difference measure used above is the difference between those distances) uses the following upper and lower bound assumptions:

(A10) The clock skew between two nodes of CLK, with respect to a given layout, is bounded above by $g(s)$ where g is some monotonically increasing function and s is the (physical) length of the path on CLK that connects the two nodes.

(A11) The clock skew between two nodes of CLK, with respect to a given layout, is bounded below by $\beta \cdot s$ where $\beta > 0$ is some constant and s is the (physical) length of the path on CLK that connects the two nodes.

Figure 3-2 illustrates these assumptions; here both the upper and lower bounds on the skew between the two communicating cells depend on the entire length of the path between them, which is the sum of their distances to their nearest common ancestor in the tree.

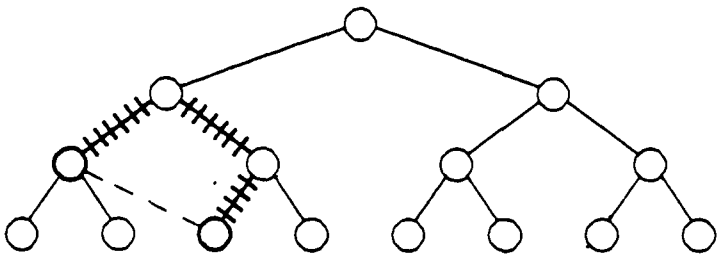


Figure 3-2: Skew in the summation model.

The two models of clock skew introduced above can be formally derived as follows, for the case when both functions f and g are linear. Let h_1 and h_2, with $h_1 \geq h_2$, be the distances of any two cells to their nearest common ancestor in the clock tree. Let $m+\epsilon$ and $m-\epsilon$ be the maximum and minimum time, respectively, to transmit a clock signal across a wire of unit length, where ϵ corresponds to the variations in electrical characteristics along clock lines. Then the clock skew σ between the two cells can be as large as

$$\sigma = h_1(m+\epsilon) - h_2(m-\epsilon) = (h_1 - h_2)m + (h_1 + h_2)\epsilon.$$

Noticing that $d = h_1 - h_2$, $s = h_1 + h_2$, and $s \geq d \geq 0$, we have

$$(m+\epsilon)\cdot s \geq \sigma = m \cdot d + \epsilon \cdot s \geq \epsilon \cdot s.$$

We see that the upper and lower bounds correspond directly to assumptions (A10) and (A11) used in the summation model, while the difference model covers the case when terms involving ϵ can be ignored.

3.4. Clocking under the difference model

Assuming the basic model defined in Section 3.2, along with condition (A9), which states that the skew between two cells is bounded above by a function of the difference between their distances from the root, we note that only a bounded amount of clock skew will occur if we ensure that all nodes in COMM are equidistant (with respect to the clock layout) from the root of CLK. This can be achieved for any layout for COMM of bounded aspect ratio, without increasing the area of the layout by more than a small constant factor, by distributing the clock through an H-tree [38]. This scheme is illustrated for linear, square, and hexagonal arrays in Figure 3-3, in which heavy lines represent clock edges and thin lines represent communication edges.

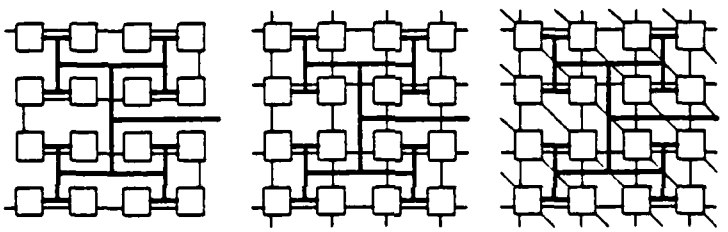


Figure 3-3: H-tree layouts for clocking (a) linear arrays, (b) square arrays, and (c) hexagonal arrays.

More precisely, we have the following result:

> **Lemma 1:** For any given layout of bounded aspect ratio, it is possible to run a clock tree such that all nodes in the original layout are equidistant (with respect to the clock tree) from the root of the tree, and the clock tree takes an area no more than a constant times the area of the original layout.

By a theoretical result [1] that any rectangular grid (for example, an $n^{2/3} \times n^{1/3}$ grid) can be embedded in a square grid by stretching the edges and the area of the source grid by at most a constant factor, we have the following theorem:

> **Theorem 2:** Under the difference model of clock skew, any ideally synchronized processor array with computation and communication delay δ bounded by a constant can be simulated by a corresponding clocked system operating with a clock period independent of the size of the array, with no more than a constant factor increase in layout area.

3.5. Clocking under the summation model

This section relaxes the assumption of the previous section by using the summation model rather than the difference model for clock skews. The clock skew between two nodes of CLK, with respect to a given layout, is related to the (physical) length of the path on CLK that connects the two nodes. Note that because the summation model is weaker than the difference model, any clocking scheme working under the summation model must also work under the difference model. The reverse of the statement is not true, however. For example, the clocking scheme illustrated in Figure 3-3(a) for linear arrays may not work under the summation model, since two communicating cells (such as the two middle cells on the left side of the layout) could be connected by a path on CLK whose length can be arbitrarily large as the size of the array grows. In the following we give another clocking scheme for linear arrays that works even under the summation model for clock skew; in addition, we show that it is impossible, under this model, to clock a two-dimensional array in time independent of its size. In this sense, linear arrays are especially suitable for clocked implementation.

3.5.1. Clocking one-dimensional processor arrays

Given any ideally synchronized one-dimensional array (Figure 3-4 (a)), we propose a corresponding clocked array (Figure 3-4 (b)) obtained by running a clock wire along the length of the one-dimensional array.

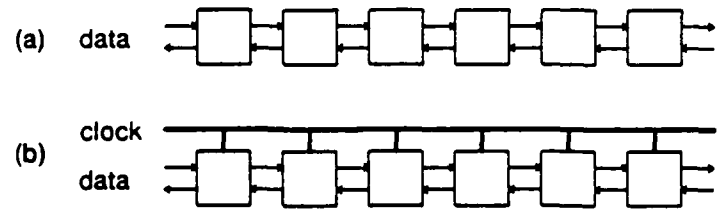


Figure 3-4: (a) Ideally synchronized one-dimensional array and (b) corresponding clocked array.

By (A10) the maximum clock skew between any two neighbors is bounded above by a constant $g(s)$, where s is the center-to-center distance between neighboring cells. Thus we have the following result:

Theorem 3: Under the summation model of clock skew, any ideally

synchronized one-dimensional processor array with computation and communication delay δ bounded by a constant can be simulated by a corresponding clocked system, as illustrated in Figure 3-4, operating at a clock period independent of the size of the array.

Skew between the host and the ends of the array can be handled similarly by folding the array in the middle (Figure 3-5), and the array can be laid out with any desired aspect ratio by using a comb-shaped layout (Figure 3-6).

With the clocking schemes illustrated, we see that the clock period for any one-dimensional processor array can be made independent of the size of the array. As a result, the clocked array may be extended to contain any number of cells using the same clocked cell design. These clocked schemes are probably the most suitable for synchronizing one-dimensional arrays due to their simplicity, modularity and expandability. Note that one-dimensional arrays are especially important in practice because of their wide applicabilities and their bounded I/O requirements [26].

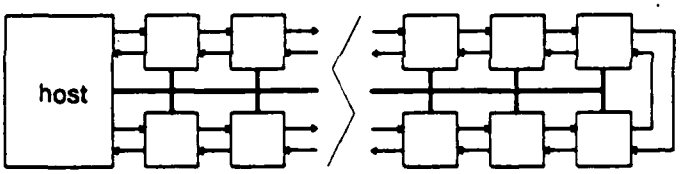


Figure 3-5: Array folded to bound skew with host.

3.5.2. A lower bound result on clock skew

We show here that the result of Theorem 3 for the one-dimensional array cannot be extended to two-dimensional structures. Consider any layout of an $n \times n$ array clocked by a global clock tree CLK; the nodes of CLK include all cells of the array. Let σ be the *maximum clock skew* between two communicating cells of the array. We want to prove that σ can not be bounded above by any constant independent of n. We use the following well known result [34]:

> **Lemma 4:** To bisect an $n \times n$ mesh-connected graph at least $c \cdot n$ edges have to be removed, where $c > 0$ is a constant independent of n.

Bisecting a graph means partitioning the graph into two subgraphs, each containing about half of the nodes of the original graph. Here for the $n \times n$ mesh-connected graph we assume that none of the subgraphs contain more than $(23/30) \cdot n^2$ nodes.

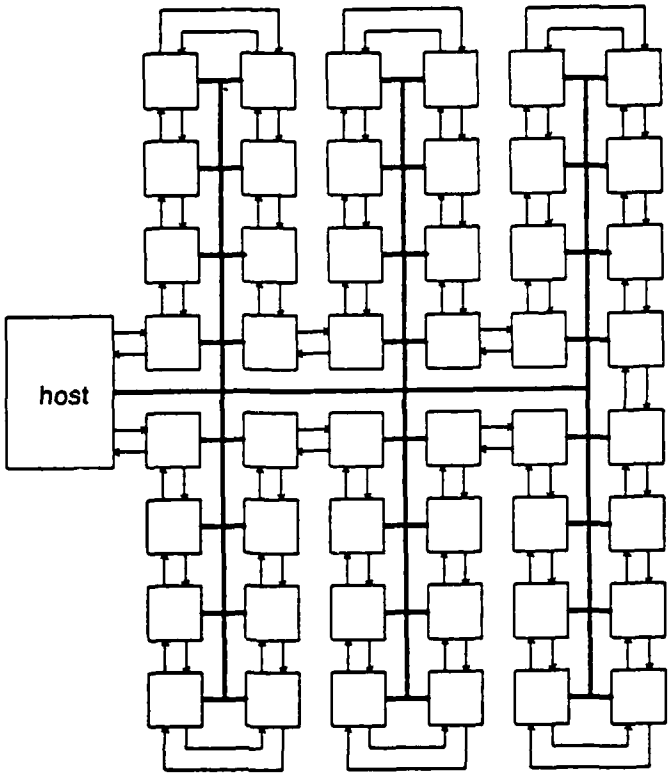


Figure 3-6: Comb layout of a one-dimensional array.

(The choice of 23/30 simplifies the proof.) We also use the following simple lemma without giving a proof.

> **Lemma 5:** For any subset M of at least two nodes of a binary tree, there exists an edge of the tree such that its removal from the tree will result in two disjoint subtrees, each having no more than two-thirds of the nodes in M.

The n^2 cells of the $n \times n$ array form a subset of nodes of CLK. By Lemma 5 we know that by removing a single edge, CLK can be partitioned into two disjoint subtrees such that each subtree has no more than $(2/3) \cdot n^2$ cells. Denote by A and B the sets of cells in the two subtrees. Let u be the root of the subtree that contains cells in A. Consider the circle centered at u and with radius σ/β, where β is defined in (A11) (Figure 3-7(a)). If there are $\geq (1/10) \cdot n^2$ cells inside the circle, then by (A2)

$$\pi(\sigma/\beta)^2 \geq n^2/10, \quad \text{or } \sigma = \Omega(n),$$

and thus σ cannot be bounded above by any constant independent of n. Suppose now that there are fewer than $(1/10) \cdot n^2$ cells inside the circle. Note that any of those

cells in A which are outside the circle cannot reach any cell in B by a path on CLK with (physical) length $\leq \sigma/\beta$. Thus these cells cannot have any communicating cells in B (with respect to the $n \times n$ array), since by (A11) the clock skew between these cells and any cell in B would be greater than $\beta \cdot \sigma/\beta = \sigma$, and the clock skew between any two neighboring cells is assumed to be no more than σ. Now let $\bar{A}$ be the union of A and the set of cells in the circle, and $\bar{B}$ be B minus the set of cells in the circle, as in Figure 3-7(b). Then $\bar{A}$ and $\bar{B}$ form a partition of the $n \times n$ array, and each of them has no more than $(1/10) \cdot n^2 + (2/3) \cdot n^2 = (23/30) \cdot n^2$ cells. From Figure 3-7(b), we see that any edge in the $n \times n$ array connecting a cell in $\bar{A}$ and a cell in $\bar{B}$ must cross the boundary of the circle. Since the length of the boundary is $2\pi\sigma/\beta$, by (A3) $\bar{A}$ and $\bar{B}$ are connected by no more than $2\pi\sigma/\beta$ edges. By Lemma 4 we have $2\pi\sigma/\beta \geq c \cdot n$, or

$$\sigma = \Omega(n).$$

Therefore as n increases, σ grows at least at the rate of n; we see that it is impossible to run a global clock for the $n \times n$ array such that the maximum clock skew σ between communicating cells will be bounded above by a constant, independent of n.

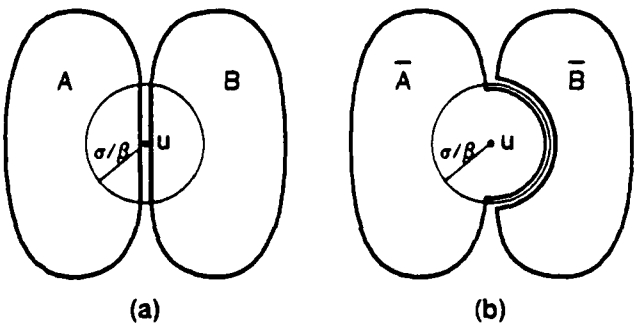


Figure 3-7: (a) original partition and
(b) new partition of the communication graph.

The above proof for two-dimensional mesh graphs can be generalized to deal with other classes of graphs. For the generalization, we need to define the *minimum bisection width* of a graph [49], which is the number of edge cuts needed to bisect the graph. For example, by Lemma 4 the minimum bisection width of an $n \times n$ mesh-connected graph is $\theta(n)$. We have the following general result:

Theorem 6: Suppose that the minimum bisection width of an N-node graph is $\Omega(W(N))$ and $W(N) = O(\sqrt{N})$. Then

$$\sigma = \Omega(W(N)).$$

Since under the summation model of clock skew two-dimensional $n \times n$ processor arrays cannot be efficiently implemented by clocked controls, their implementation should be assisted by some self-timed scheme as discussed in the next section.

3.6. Hybrid synchronization

In the absence of the invariance condition (A8), in which case pipelined clocking fails, or for communication graphs with asymptotically growing clock skews under the summation model, global clocking is unable to provide constant clock rates as a system grows. In this case, an scheme similar to one described by Seitz [44], where local clocks are controlled by a self-timed handshaking synchronization network, can be used.

In this approach, we break up the layout into bounded-size segments called *elements*, and provide each element with a local clock distribution node. The clock distribution nodes employ a handshaking protocol to synchronize among themselves, and then distribute clock signals to the cells in their elements. Given assumptions about the maximum delay of a computation node and its communication wires, we can clock the cells in each element in constant time. This structure is illustrated in Figure 3-8, in which the heavy lines and black boxes represent the self-timed synchronization network, and the narrow lines represent local clock distribution to the cells in each element. Note that the subordination of the local clocks to the self-timed network avoids the possibility of synchronization failure due to a flip-flop entering a metastable state, since an element stops its clock synchronously and has its clock started asynchronously.

This provides the asymptotic performance of a self-timed system by making all synchronization paths local, while isolating the self-timed logic to a small subsystem and allowing the cells to be designed as if the entire system were globally clocked. The hybrid approach has the additional advantage that a single synchronization

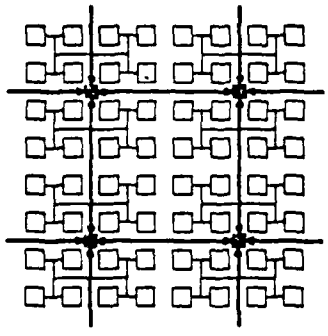


Figure 3-8: Hybrid synchronization scheme.

design can be used for many different structures. This simplification of the usual fully self-timed scheme is made possible by the fact that we are willing to assume a maximum delay for the cells; this is the same assumption made in ordinary clocked schemes. Note that we are willing to let the entire array operate at worst-case cell speed, since even a fully self-timed array would usually wind up operating at that speed regardless.

3.7. Practical implications

This section addresses some practical aspects of the material of this chapter. Perhaps the most important practical issue raised in this study is the question of the use of pipelined clocking on chips; this section discusses some of the potential limits to this technique, and presents some simple experimental evidence suggesting its practicability. It also discusses the practical relevance of the difference and summation models of clock skew.

The practicality of pipelined clocking hinges on two issues: the limitations of the uniformity assumption (A7), and the delay associated with distributing clocks in a conventional fashion. Pipelined clocking only makes sense if clock event transmission is uniform enough to gain an advantage over equipotential clocking.

This relationship, in turn, depends on the relative speeds of logic and interconnect. Pipelined clock trees, with short interconnection paths, will run at logic switching speeds (to the extent that uniformity obtains.) The speed at which equipotential clock trees can run is determined by the impedance of the interconnect and by its

physical dimensions. We would thus expect pipelined clocking to be most applicable where switching speeds are high and interconnect is long and has high impedance; for example, wafer-scale gallium arsenide may be a likely candidate.

The uniformity of transmission of clock events is subject to a number of factors. One obvious limitation is the uniformity of a buffer in passing rising and falling edges. For an nMOS superbuffer, for example, making transit times for rising and falling edges the same requires careful circuit tuning, and the resulting circuit will be very sensitive to manufacturing process parameters. One solution to this problem is to make each buffer respond only to rising edges on its input, and generate its own falling edges with a one-shot pulse generator. This solution has the disadvantage that the pulse width must be either wired into the circuit or programmable by some means. This may actually be convenient in some cases, if the pulse generating circuitry can be designed to model the delay of the logic circuitry. Beyond static considerations, however, the transmission of clock events can also be affected by noise, for example internet capacitive coupling in MOS circuits. This problem can only be avoided by careful design, and further research is needed in estimating its magnitude.

Another, simpler approach to the rising/falling edge problem is to build a distribution line as a string of inverters. If the impedance of the outputs of the odd inverters is the same as that of the even inverters, rising and falling edges should traverse the string at essentially the same speed. Although this approach eliminates any inherent bias in favor of one type of edge, it does not result in speed independent of the length of the inverter string. Assume that the discrepancy between rising and falling transit times for a pair of inverters is normally distributed with a mean of zero and variance V. The sum of the discrepancies of n inverter pairs will be similarly distributed, with variance nV. If a fixed yield, independent of n is desired, chips with a discrepancy sum proportional to the standard deviation, hence proportional to $\sqrt{n}$, must be accepted. Since a minimum condition for a given chip to run with cycle time T is that the sum of discrepancies be no greater than T, some chips will run with cycle times at least proportional to $\sqrt{n}$. The constant factors involved may still be small enough, however, to make this scheme feasible in practice.

As a simple trial of practical issues, an nMOS chip consisting of a string of 2048 minimum inverters was designed, without any special attention paid to making interconnect impedance uniform. An equipotential single phase clock signal could be run through the entire string with a cycle time of approximately 34 μs; even with the disadvantage of a slight bias in the circuit design toward falling edges, a pipelined clock could be run with a cycle time of 500 ns, 68 times faster. The same speedup was observed on five separate chips, indicating that the effect of the bias in the circuit design dominated the type of probabilistic effects described above. Assuming that transit times and any discrepancy between rising and falling edges scale linearly, a similar inverter string of any length could be clocked 68 times faster in pipeline mode than in equipotential mode. This figure does not indicate that pipelined clocking is actually applicable in this case; a chip of this size in this technology could easily be clocked with a 50 ns cycle time with a well-designed, low-resistance equipotential clock. However, it does suggest that pipelined clocking may well be feasible where switches are fast and wires are slow.

The second practical issue related to this work is the question of the applicability of the difference and summation models of clock skew. First, both models apply only where pipelined clock distribution is possible; otherwise, clock period inevitably grows with the system and the only means of improving performance are technology improvement, clever design, and self-timing. For the difference model to apply and for H-tree or other equidistant clocking schemes to be useful, it must be possible to closely control the "length" (that is, the delay characteristics) of the clock tree. This is possible in systems where wires are discrete entities that can be tuned, and indeed this is common practice in such systems. Whether this is true for integrated circuits is another question, hinging on the variability of the fabrication process and on noise characteristics.

The summation model is much more robust. Given the possibility of pipelined clocking, almost any imaginable means of transmitting clock events will have the property that cells close together on the clock tree will have bounded skew between them. We can thus be confident that linear arrays and similar structures will work as well as pipelined clocking can work.

3.8. Concluding remarks

In this chapter, we have analyzed the effect of clocked synchronization on the performance of large processor arrays. We have identified the key issues on which this depends (clock delay and clock skew), and proposed means of implementing pipelined clocking for integrated circuits. We have considered two models of the dependency of clock skew on layout properties, and derived upper and lower bounds for the performance of processor arrays of varying topologies. The key results here are that one-dimensional arrays can be clocked at a rate independent of their size under fairly robust assumptions, while two-dimensional arrays and other graphs with similar properties cannot. We have also discussed some of the practical implications of these theoretical results.

This study has concentrated on the interaction of clock skew models with the communication structure of arrays with bounded communication delay; future work should also examine cases where asymptotically growing delays occur. One interesting such case is that where the communication graph COMM, neglecting edge directions, is a binary tree. It has been shown that a planar layout of a tree with N nodes of unit area must have an edge of length $\Omega(\sqrt{N}\ /\log N)$ [41]. Under the summation model of Section 3.5, then, if we make the additional assumption that communication delays grow with path length in the same way as clocking delays, a tree may be clocked at no loss in asymptotic performance simply by distributing clock events along the data paths.

Furthermore, if COMM is acyclic, as in the tree machine algorithms described in a paper by Bentley and Kung [6], and the ratio between lengths (in the layout) of any two edges at the same level in the graph is bounded, pipeline registers can be added on the long edges, with the same number of registers on all of the edges in a given level. This makes all wires have bounded length, thus causing the time needed for a cell to operate and pass on its results to be independent of the size of the tree. Adding the registers increases the layout area by at most a constant factor, since they in effect just make wires thicker. For example, an H-tree layout has this property, and allows a tree machine of N nodes to be laid out in area $O(N)$ with delay through the tree of $O(\sqrt{N})$ and constant pipeline interval.

4 Serialized Implementations

Bit-serial implementations of systolic algorithms are attractive for reasons of ease of design, high clock rates and flexible word length. Their ease of design arises from their construction from many bit-serial arithmetic parts, which are small and are replicated many times, giving the design a high regularity factor. Clock rates are potentially higher than in systems using parallel arithmetic, as no provision need be made for carry propagation or broadcast of operands (as, for example, in a parallel multiplier). The systems of interest in this chapter are not those which use serial arithmetic and I/O to reduce hardware and pincount at the expense of throughput, but rather those which aim to increase throughput by using many serial cells in parallel at high clock rates.

This chapter deals with two aspects of serial implementations: their design and their cost and performance. In the first section, we show how serial arrays can be derived, in two steps, from word-parallel arrays of inner product cells. We also explain the derivation of two previously described bit-serial convolution arrays [12, 13] in these terms. In the second section, we consider the costs and benefits of serial arrays, and list some factors which determine the optimal degree of serialization.

4.1. Serializing word-parallel arrays

Two transformations on combinational logic blocks are fundamental in deriving bit-serial arrays from word-parallel arrays. The first involves serializing computations in order to reduce the amount of combinational logic used. The second uses pipelining to reduce the cycle time of a machine while retaining the original per-cycle throughput.

In the context of this chapter, serialization means breaking up a computation which is performed combinationally into identical steps, which are then scheduled serially on a single, smaller piece of logic. Thus serialization is essentially a hardware-saving operation. A computation will take more steps than the original parallel implementation, but the time to compute a step is typically smaller. An example is the case of integer addition, where the avoidance of carry propagation allows a serial implementation to have a shorter cycle time than a parallel one. Note that parallel words can be serialized into chunks of any length, not just single bits.

For our purposes, pipelining refers to breaking up a computation into stages which need not be identical, but for greatest efficiency have identical delays. All of the original logic is used, but delay registers (latches) are introduced between the stages so that a new problem can enter each stage as soon as the previous problem has completed that stage. Here the throughput in terms of results per cycle remains the same, but the amount of time needed to perform each cycle is decreased. In order for simple pipelining to work, individual problem instances cannot depend on the results of closely preceding instances, since those results will not be ready until the entire pipeline has been traversed. For example, pipelining of cell computations in systolic arrays can cause scheduling difficulties in algorithms with feedback [28].

4.1.1. An example

We can take an array of inner-product cells, described in terms of parallel multiplications, and turn it into a serial array in two steps. First, we replace each multiplier with a parallel adder which is used repeatedly to perform the multiplication. This is a serialization transformation. This array has the same timing as before, in the sense that if we view each adder and its associated registers as an inner-

product cell, the only difference is that multiplication is slower by a factor of the length of a word. Note also that we can take advantage of the slowdown by serializing intercell communication without changing any timing relationships.

Next, we pipeline the parallel adders, in effect turning them into serial multipliers. The result is a serial array. An optimal choice of how the multiplication is serialized and how the addition is pipelined will depend on the word-level data flow, but will have only small constant-factor effects. This pipelining step is not affected by the usual restriction on pipelining mentioned above; since we have serialized the original system by the same degree, we are now assured of a sufficient supply of problems to keep the pipeline full, regardless of feedback in the original array.

As an example, consider a standard convolution array [26] where weights are stored in the cells and the input and result streams flow in opposite directions at equal speed, as shown in Figure 4-1(a).

Serializing the inner product logic turns the multiplier into a shifting accumulator built around a parallel adder, and the adder into a serial adder. In each cycle, the stored weight is conditionally added to the shifted accumulator.

As noted above, slowing down the multiplication also allows data transfer to be serialized; hence a shift register is supplied for the input stream. The result stream is generated online with only unit delay at each cell: in each cell, the incoming result bit is added to the least significant bit of the accumulator. The low-order sum bit is sent out as the outgoing result, and the carry is retained in the serial adder for the next cycle.

This serial cell is shown in Figure 4-1(b). For b-bit data and weights, the x bit sequences are separated by $b-1+k$ zeroes, $k \geq \lceil \log n \rceil$, to allow for the accumulation of n terms of up to $2b-1$ bits. (Alternatively, a major/minor clocking scheme could be used, where the x's would not move for $b-1+k$ cycles. This would save register cells, but complicate timing and control.) Each of the x shift registers has $2b-2+k$ cells, so that the least significant bit of x_i is used one cycle before the corresponding bit of x_{i+1} is used in the cell to the left. This assures that the y sequence meets bits of the appropriate significance.

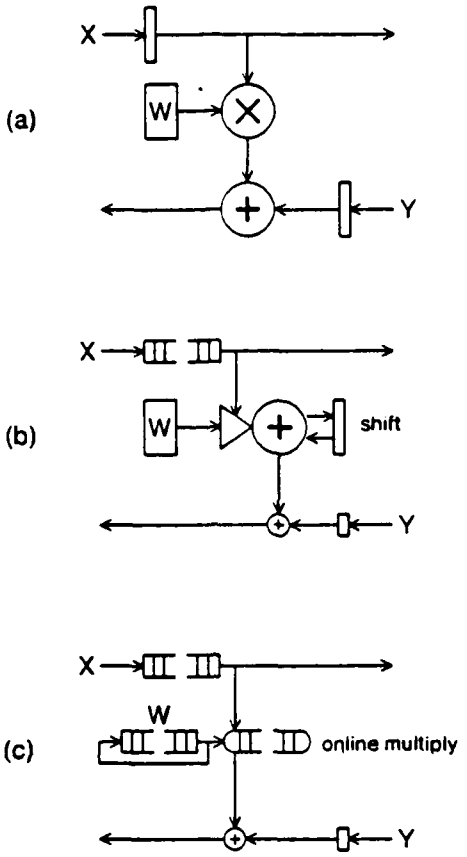


Figure 4-1: (a) Parallel convolver cell, (b) serialized version and (c) pipelined serial version.

In the pipelining stage, the parallel adder can be pipelined and incorporated with shifting logic to form a serial multiplier [3], as in Figure 4-1(c). This scheme results in the same logical I/O schedule, but with a smaller cycle time.

4.1.2. Analysis of existing designs

Evans *et al.* have reported a bit-serial systolic convolver [13], designed without reference to a bit-parallel algorithm. The convolver, as shown in Figure 4-2, consists of a stack of serial multipliers, where partial products are accumulated by being sent downward through the array. Input data pass serially left to right and upwards in the array, with the rightmost cell in each row passing its input bit to the leftmost cell of the next higher row. Coefficients circulate serially in each row, one coefficient per row.

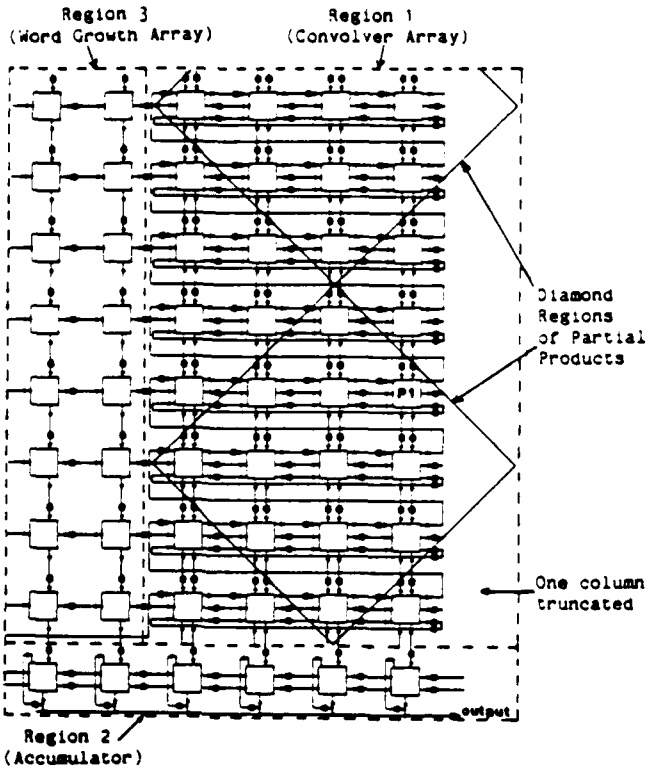


Figure 4-2: Convolver of Evans, *et al.* [13].

From the transformational point of view, this algorithm can be seen as a transformed instance of the familiar convolution array of Figure 4-1(a). First, the array is serialized in a fashion dual to that shown in Figure 4-1(b): instead of keeping the x's in shift registers and processing the y's online, we keep the y's in shifting accumulators and delay each x bit by only one clock cycle in each cell. In pipelining the conditional adder, we then use the serial multiplier shown in Figure 4-3. Here the weight bits are shifted left to right, least significant bit first, and the x bits are shifted right to left, most significant bit first. Each cell of the multiplier has a fixed significance, so all partial product bits of a particular significance are generated in the same cell.

The remaining operation needed is the accumulation of the results. One way to handle this is to keep the result in the multiplier cells, accumulating partial product bits, and then to add it into the multiplier of the next cell as an external set of partial

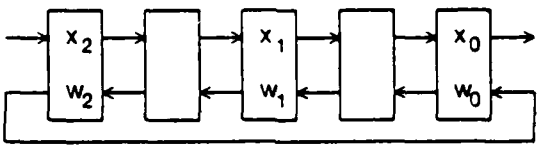


Figure 4-3: Serial multiplier.

products when all terms are collected. The alternative adopted by Evans *et al.*, which has the virtue of avoiding major/minor clock cycles, is to pass each set of partial products to the multiplier below as it is computed. Final accumulation of all partial products and propagation of carries takes place at the bottom of the array. The only remaining feature of the array to be derived is that Evans *et al.* delete the least significant bit of each multiplier to save hardware; the bit multipliers are absent, but the corresponding delay registers remain.

Corry and Patel have reported a different bit-serial convolver [12] in which each array handles only one-bit weights, and several chips are stacked to use larger weights. The structure of the convolver with one-bit weights is shown in Figure 4-4. The input data enters the array from the left, with the least significant bit at the top, and skewed so that the bits of one input word enter in successive cycles, least significant bit first. The results travel in a similar pattern, headed to the left. Carries are passed downward in the array. Weights ("reference" values in the figure) move downward, so that they can be changed for adaptive filtering.

This design too can be derived by transformation from a word-level design. The preliminary decomposition of the convolution weights eliminates the serialization step, as multiplication is reduced to conditional addition. The design of Figure 4-1(a) can also be used here (assuming constant weights; adaptive weights are easily added) — the only change is that the multiplication becomes a conditional addition to y. The adders are pipelined in the same fashion as the comparators in Foster and Kung's pattern matching algorithm [18]: each result is computed least significant bit first, and carries are sent to the next more significant position for the next cycle. This imposes the time skew seen in Figure 4-4.

The two-step procedure of serializing and then pipelining can be applied to any data-independent systolic algorithm using parallel multipliers. Data-independent

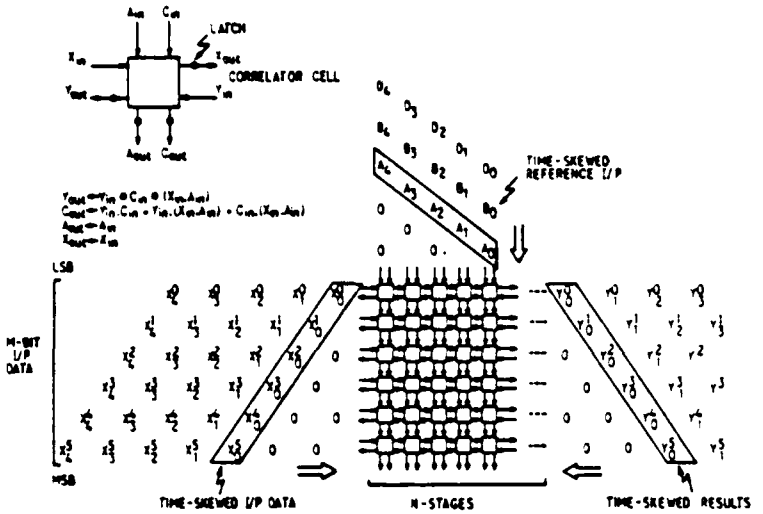


Figure 4-4: Convolver of Corry and Patel [12].

algorithms using parallel adders or comparators can be pipelined without first undergoing the serialization step. It is worth noting that there is little hardware savings to be had by serializing a parallel adder or comparator, as storage for an entire word of data is required by most algorithms in any case.

4.2. Cost and performance

Bit-serial implementations have two major attractions: simplicity of design and small cycle time. Design is simplified because there is no need for carry trees (as in parallel adders) or for global buses (as in parallel multipliers). Cycle time is smaller because the critical path of the computation is essentially just a single bit operation. These advantages are countered by additional area required for latches and larger clock drivers (for higher speed and more clock capacitance). Also to be taken into account are the area and delays of other components of the system. This section discusses some of these issues, and points out that in many cases, multiple-bit operations may be competitive or better. Note that our attention here is directed not to systems where serialization is adopted for small area, but rather where it is paired with pipelining or parallelism for increased throughput.

4.2.1. Area considerations

Bit-serial implementations incur a larger proportional amount of overhead for non-computational circuitry than do parallel implementations. This overhead may have several different sources, depending on the structure of the array.

The largest and most technology-independent effect is the area devoted to latches. A bit multiplier in a fully serialized system will normally have four associated latches (two operand bits, a carry bit and a partial product bit). Since an adder itself does not require a great deal of circuitry, the latches may double or triple the area of the cell, depending on technology and the type of latches used.

Another consideration, not found in all systems, is the balance of control circuitry with data-handling circuitry. Control circuitry may range from none, to one bit per cell (as in Evans, *et al.*'s convolver [13]), to the execution logic of a SIMD processor, to the extreme found in MIMD bit-serial processors. As the non-numeric logic in a cell expands, it becomes increasingly germane to ask by what proportion the cell would grow if, for example, two-bit arithmetic were used rather than one-bit. If, for example, a cell's area would increase by 20% and its cycle time by a factor of 2, the fourfold improvement in the number of cycles needed to perform a large multiplication might make the substitution worthwhile. A related question is the interconnection topology of the cells. If connections are nearest-neighbor, the area overhead of interconnection is small; if a topology with a nonlinear area growth rate is used, the large number of cells in a bit-serial system may make it uneconomical.

Yet another factor is the area and complexity cost of rapidly clocking many latches. For MOS implementation, clock drivers may need to be quite large: for example, the HP-9000 microprocessor chipset devotes most of one chip to clock drivers whose output transistors are 55,000 microns wide by 2.1 microns long and produce peak currents of two to three amperes [35]. In general, the ratio between the area needed to drive the clocks of a serialized array and that needed to drive the clocks of a parallel array will be similar to the product of the ratio of the number of latches (greater capacitance requires larger drivers, and gate capacitance dominates interconnect) and the ratio of clock frequencies (a smaller rise time requires larger drivers). Since the main advantage of pipelined systems is that clock frequencies can

increase proportionally with the number of pipeline stages, the area cost of clocking a serialized array can be quadratic in the number of pipeline stages.

4.2.2. Speed considerations

The speed at which a chip operates in a system is dependent not only on the critical path of gate delays in its major computational circuitry, but also on constraints imposed by other circuits on the chip, synchronization requirements and the outside world.

The delay of circuits other than those performing bit arithmetic becomes more significant as a system grows more complex. A system incorporating, for example, a barrel shifter or conditional control elements will not be able to run as fast as the bare bit-serial array.

Internal clock skew and rise and fall times always present a lower bound for chip clock periods. When the combinational delay of each stage is very small, these times may represent a significant overhead. The clocking problem can become even more serious in systems using a one-phase clock, where combinational delay is expected to prevent flip-flop hold time violations; if the logic is too fast, synchronization errors may occur. The overhead argument also applies to self-timed systems; the time needed to perform a handshaking transaction will often dominate the time taken by a bit operation.

The ability of a chip to transmit its results to the outside world also becomes an issue at very high clock rates. For MOS circuits, a fairly large number of gate delays is needed to drive package and circuit board leads, which are far more capacitive than on-chip conductors. Also, the high-speed switching of multiple pins can cause serious noise problems due to the inductance of the chip's power supply wiring.

The other side of interfacing with the outside world is the speed at which the outside world runs. System clock rates and I/O bandwidths are often bounded by technological and engineering considerations that have little to do with on-chip circuit speeds. Even when these factors are not the limit to system speed, matching a bit-serial part with a critical path of three or four gate delays to system components

with a critical path perhaps four times as long requires careful attention to timing schemes and overall I/O bandwidth.

4.2.3. Other considerations

Two other issues can, in some cases, make bit-serial arrays less attractive. These issues have to do with logic optimization and with inherent parallelism and predictability in applications.

A potential disadvantage of the bit-serial approach is that opportunities for optimization and implementation tradeoffs at the cell level are quite limited, due to the simplicity of each cell and the rather small number of possibilities for its implementation. When larger combinational units are considered, the possibility of using a faster and/or smaller algorithm (such as a carry-lookahead adder or a Booth's algorithm multiplier) arises. These approaches reduce the appeal of serial over parallel implementations. Another possibility for more parallel implementations is a tradeoff between logic depth and breadth; for example, it may be advantageous to devote extra gates to implementing a 2-bit multiplication, thereby reducing its critical path length.

The second point depends on properties of the application being served. To achieve high throughput, bit-serial systems will be either highly pipelined or highly parallel. For pipelined systems, the latency in cycles from input to output of a given bit will be long. For parallel systems, control of the operations being performed will likely be either global or predetermined, for reasons of control circuitry overhead. In either case, the predictability or inherent parallelism of the algorithm being performed must be high enough to keep the pipeline full or the processors busy. Thus, in order to realize in practice the potential throughput advantages of serial arrays, algorithms must exhibit high regularity. This may pose serious difficulties for general-purpose machines.

4.2.4. Examples

This section briefly discusses two existing bit-serial systems where area and time numbers are available. Taking some of the above considerations into account, we suggest possible improvements in area/time performance.

The systolic convolver of Evans, *et al.* [13], described earlier, is implemented as a CMOS chip with a minimum clock cycle time of 50 ns. The combinational delay of a one-bit multiplier cell, however, is given as 15 ns (presumably including latch setup and hold times). Approximately 2/3 of the area of each cell is apparently devoted to latches, which are described as relatively small.

Now consider a similar chip in which the combinational building block is a 2×2 parallel multiplier-accumulator, but the same total number of full adders is used. An upper bound on the area devoted to four full adders in the new design is the corresponding area used in the old design, less half of that portion devoted to latches. Thus the total chip area devoted to the multiplier array would shrink by about 33%, and smaller clock drivers could be used. Offsetting this area advantage would be an increase in the combinational delay of each cell (throughput per clock cycle remains the same, and latency decreases). Since the new cells should presumably have a delay of no more than 30 ns and since pipelining allows them not to reside in the chip's critical timing path, this area savings can be achieved at no cost in performance. There is, however, a price to be paid in design complexity.

A case with significantly higher overhead is MPP [5], the Massively Parallel Processor. MPP is a bit-serial SIMD multiprocessor structured as a 128×128 grid. The machine is built of high-speed CMOS chips, each of which holds eight bit-serial processing elements and runs with a cycle time of 100 ns. In this case, making each PE capable of handling two bits at a time would have little effect on total area or on cycle time, but could double (or quadruple, in the case of integer multiplication) throughput. This might not be feasible for reasons of I/O bandwidth to PE memory. Even if the number of PEs on a chip were reduced in order to make two-bit computation feasible, however, there would still be some improvement in throughput for a given hardware investment. Simplicity of programming will favor the single-bit structure in many applications, however.

5 Dictionary Machines — A Case Study

This chapter contains a case study in the design of parallel algorithms and architecture, incorporating many of the criteria needed in making implementation decisions for special-purpose systems. The application addressed is the dictionary problem: performing a group of dictionary operations (INSERT, DELETE, EXTRACTMIN, NEAR, etc.) on a set of keys. A number of tree-structured multiprocessor designs have been proposed for this problem. These designs typically use one processor for each key stored and operate with constant throughput, assuming unit time to communicate and compare keys. This assumption breaks down in applications with long keys. This chapter describes a machine which uses a number of processors proportional to the maximum length of a key to achieve constant throughput, regardless of key length. This design has important practical advantages over the family of tree-structured machines, and demonstrates that processor-intensive VLSI structures are not always the best route to a high-performance system.

5.1. Introduction

The dictionary task can be loosely defined as the problem of maintaining a set of keys drawn from some ordered domain; often some indivisible piece of information, a record or pointer to a record, is associated with each key. Maintaining the key database consists of performing a series of update and query operations on its contents. A typical set of operations includes some subset of the following:

INSERT add a key to the database.

DELETE	remove a key.
FIND	determine whether a key belongs to the database (and return its associated information).
EXTRACTMIN	remove and report the lowest key in the database.
EXTRACTMAX	remove and report the highest key in the database. (Some data structures can support EXTRACTMIN or EXTRACTMAX, but not both simultaneously.)
NEAR	report the stored key closest in the domain ordering to a specified query key.

The INSERT and DELETE operations come in two flavors: redundant and non-redundant. An insertion is redundant when the key being inserted already exists in the database; a deletion is redundant when the key being deleted does *not* exist. Allowing redundant operations is more natural in some applications than forbidding them (for example, a user might wish to insert a key without knowing or checking whether it already exists), but can cause difficulty in pipelined implementations.

A great number of useful computations can be cast in terms of the dictionary task. The simplest example is a symbol table (or dictionary), where only the INSERT, DELETE, and FIND operations are used. Another example is priority queues, which use INSERT and EXTRACTMIN (and EXTRACTMAX, if the queue is actually a deque). A median filtering algorithm could use INSERT, DELETE, and "EXTRACTMIDDLE", which can be implemented by using the other EXTRACT operations on a bisected database. Pattern matching systems might use INSERT, DELETE, and NEAR.

In general, the best implementation of the dictionary operations for a particular application will depend on the number and size of the keys to be stored, on the particular set of operations to be performed, and on the cost/performance goals of the system. This study first reviews a few serial algorithms for some versions of the dictionary problem. It then surveys a family of tree-structured algorithmic machines [2, 32, 40, 45] which have been proposed as high-performance solutions, along with a tree algorithm which runs on a linear array of processors [10]. Finally, it proposes a new parallel algorithm, based on a linear array of a small number of processors with large memories, which has important advantages over the tree architectures and is preferable to the previous linear array architecture in many cases.

Details of the tree machines, a cost estimate for the new algorithm, and an asymptotic analysis of the various algorithms are described in Appendices 5.A, 5.B and 5.C.

A note on notation: Following Ottmann *et al.* [40], the variable n is used to denote the number of keys stored in the database at a particular time, and N to denote the maximum number of keys that may be stored. Two types of time complexity are considered: *latency*, the time elapsed between the initiation and completion of a query, and *pipeline period*, the minimum time between the initiation of two separate operations. The pipeline period of an algorithm is inversely proportional to its throughput. We will also distinguish, in complexity expressions, between *key* complexity (storing or operating on entire keys) and *symbol* complexity (dealing with symbols drawn from a finite alphabet). This distinction is needed to compare algorithms which treat keys as unit objects with those that treat keys as strings of symbols. Complexity functions in each case are written $O_k(f)$ and $O_s(f)$, respectively. The function "log" represents the base 2 logarithm.

5.2. Previous solutions

This section briefly reviews two types of realizations of the dictionary task: serial algorithms and parallel algorithmic architectures. Four serial algorithms are illustrative in this context. Each is informally summarized here; details can be found in Knuth [23]. Also briefly surveyed are a family of tree-structured parallel architectures and a linearly structured architecture; more details on the tree machines are given in Appendix 5.A.

Hashing Hashing algorithms store keys in a table, with indices derived from the keys by a random or pseudorandom hash function. Collisions are resolved by one of a number of methods. Assuming a uniform distribution of hash values and a table that is not too full, the expected time to perform INSERT, DELETE, or FIND is $O_k(1)$, though this grows in the worst case to $O_k(n)$. The algorithm uses $O_k(N)$ space, though this can be reduced to $O_k(n)$ at the cost of occasional reorganization and $O_k(n)$ worst-case insertion time by doubling the size of the table and rehashing each key when the table grows too full.

Heaps A heap is a binary tree of keys organized so that all of a key's descendants have higher values. The worst case time to execute INSERT or EXTRACTMIN is $O_k(\log n)$, and the algorithm uses $O_k(n)$ space. Because of the weak ordering conditions, the tree is always balanced and can be implemented without pointers.

Balanced trees In this scheme, a binary tree of keys is maintained so that all of a key's left-hand descendants have smaller values while all of its right-hand descendants have larger values. The tree is kept balanced by rotation operations. All of the dictionary operations can be performed in $O_k(\log n)$ time. Two auxiliary pointers are associated with each key.

Tries [radix trees] The name "trie" is derived from "information re*trie*val"; because its pronunciation either sounds like "tree" or is nonintuitive, the term "radix tree" is used here instead. Nodes in a radix tree represent prefixes of keys in the database, represented as strings over some alphabet $\sum$. The root of the tree is the null string, and a node representing substring α has a descendant corresponding to $s \in \sum$ if αs is also a substring of a key in the database. Figure 5-1 shows a small radix tree storing six words. All of the dictionary operations can be performed in $O_s(l)$ time, where l is the length of the key specified or retrieved.

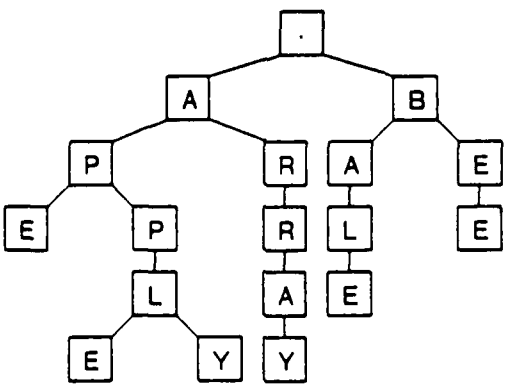


Figure 5-1: Radix tree.

Since its speed is independent of the size of the database, this method is faster than balanced trees for large collections of keys. On the other hand, radix trees require much more space for pointers: one pointer is needed for each prefix, and a factor of $|\sum|$ may be wasted in the worst case if nodes are represented as tables.

This space penalty is ameliorated by prefix sharing, since each prefix in the database occurs just once. Nodes in the tree where there is no branching offer additional opportunities for compression. Knuth mentions a few transformations that can be applied to reduce memory requirements, and many more can be imagined.

Tree architectures At least four different papers in the literature [2, 32, 40, 45] propose tree-structured architectures for the implementation of dictionary operations. While they differ in detail, they all share some basic principles. A machine is a rooted, balanced binary tree of processors, each processor holding one or a few keys. Dictionary operations are broadcast from the root, and are pipelined along the depth of the tree. Because of this pipelining, the tree structures have pipeline period $O_k(1)$, independent of n.

All of these architectures are processor-profligate in the sense that they use $\theta(N)$ processors to achieve only an $O(\log n)$ throughput improvement over the serial balanced tree algorithm. Asymptotically, this is an improvement, at least where n is not too much smaller than N. The algorithm runs faster, and the asymptotic hardware cost is the same. In the real world, though, if n keys fit in primary storage, $\log n$ is not likely to be much larger than 25 or 30, and a processor takes many more transistors to build than a few bits of memory. Another issue is that for applications where duplicate keys are not allowed or are stored as one key with a count, the length of a key in bits is at least $\log n$, and in most cases is several times larger. Thus it may be possible to break up the processing of a key over several processors, and achieve a factor of at least $\log n$ in parallelism along the length of a key.

In defense of the tree architectures, it should be noted that they offer capabilities which go beyond dictionary operations, and which require $O_k(n)$ time on a uniprocessor. These capabilities include parallel modifications of stored keys and NEAR-style searches where the distance measure is not constrained by the key ordering. Although most of the papers cited do not mention the possibility of using their designs for anything other than dictionary tasks, they are more likely to be useful in solving more difficult problems.

An $O(\log N)$-processor architecture The many processor – small speedup problem of the tree architectures has been previously addressed by Carey and

Thompson [10]. They propose a linear array of $O(\log N)$ processors, each of which stores one level of a special type of B-tree. Keys are stored at the leaves of the tree, and interior nodes hold keys which direct a search to one of two, three, or four descendants. The tree is balanced by splitting full nodes and merging nearly empty nodes. The algorithms for all of the dictionary operations can be pipelined with period $O_k(1)$.

5.3. A level-parallel radix tree algorithm

We have noted two undesirable features of the tree machine designs. The first is that they use $O(N)$ processors to achieve an $O(\log n)$ speedup over uniprocessors. The second is that, by treating keys as indivisible units, they do not do as good a job as possible on long keys. Carey and Thompson's design is also subject to the second criticism. This section proposes a parallel implementation of radix trees that addresses these problems. It first describes a basic version of the algorithm, along with a "radix machine" on which it runs, and then indicates some cost-performance tradeoffs and efficiency tunings. This is followed by a comparison with previous designs.

5.3.1. The algorithm and the radix machine

The central idea of the parallel radix tree algorithm is to attach a processor to each level of the tree, starting with the children of the root (*i. e.*, the first symbols of the keys). A machine to execute the algorithm has L processors, where L is the length of the longest key. Radix tree nodes representing prefixes of length k are stored in a local memory attached to processor k. The processors are connected in a linear array in consecutive order. The symbols of each input key arrive at the appropriate processors from the side of the array, skewed one cycle apart so that the processing of a key is pipelined along the array. Output keys are produced in a similar fashion. This organization is sketched in Figure 5-2, which shows several input keys (read diagonally) approaching from the right. Input symbols which are vertically aligned will enter their respective processors at the same time.

The data structure is just as specified above for the serial algorithm, except that a

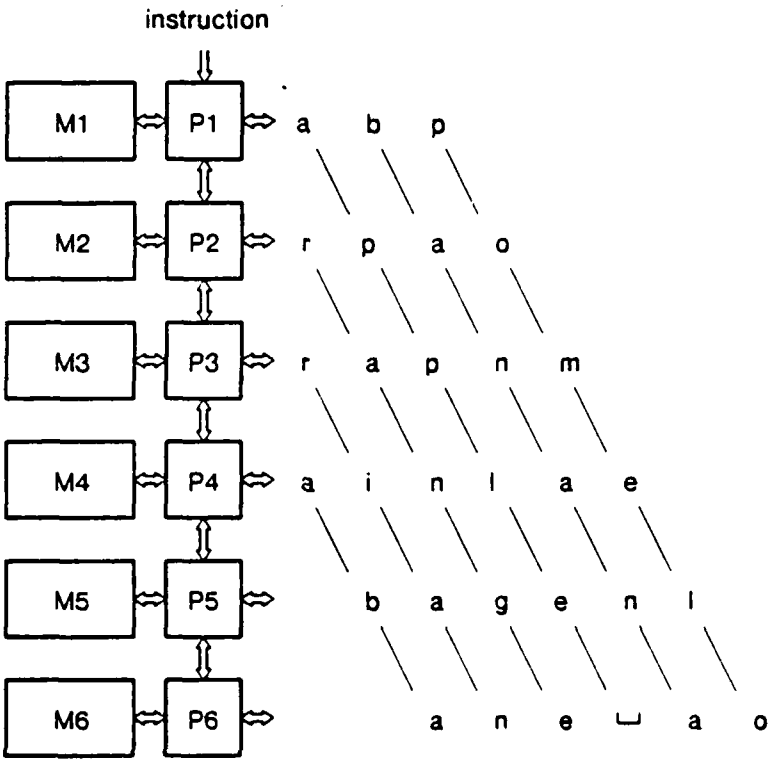


Figure 5-2: Structure of a radix machine.

descendant count, the number of keys beginning with a particular prefix, is added to each node. This count is updated along the length of each string that is inserted, deleted, or extracted. The count is used to allow pipelining of operations that remove keys from the database.

The procedures for carrying out each of the operations are similar to the serial case, except for some changes that must be made to accommodate pipelining. If redundant INSERTs or DELETEs are not allowed, then every operation proceeds from root to leaf. DELETEs and EXTRACTs return nodes to a free list in each memory as their descendant counts become zero, and INSERTs allocate new nodes as necessary.

If redundant operations are allowed, a scheme similar to that used by Atallah and Kosaraju [2] is employed. In this approach, insertions are performed as if nonredundant, and if necessary a correction pass is performed from the bottom to the top of the tree to decrement spuriously incremented counters. Deletions are assumed to be redundant, and no action is taken on the trip down the tree. If the deletion turns out

to be valid, an upward pass is taken which performs the appropriate decrementing of counts and freeing of nodes. Upward passes which begin in processors other than the last are delayed so that a "round trip" always takes $2L$ cycles; thus at most one upward pass is active in a single processor at any time, and it is always possible to tell whether a deletion or insertion is valid when the end of a suffix is reached. Note that this scheme prevents erroneous deletions; the descendant count stored at a node is always greater than or equal to the eventual "correct" value. Note also that the information necessary for upward trips, except for a single bit traveling upward indicating redundancy or nonredundancy, can be stored in a small ring buffer at each processor.

This scheme limits the pipelining of operations in the following way. An EXTRACT or a NEAR following within $2L$ cycles of a redundant INSERT or nonredundant DELETE can be "fooled" down the wrong path by counts which are erroneously high. An EXTRACT or NEAR is on the wrong path if it is at a node with only one possible suffix, and that suffix is about to be deleted by a correction pass which is delayed or moving upward. If that is the case, the EXTRACT operation will delete nodes as it moves downward, and the correction pass will delete nodes as it moves upward. When they meet, the correction pass will stop, since the EXTRACT has already done its work higher up in the tree, and the EXTRACT will report an error. In the case of a NEAR operation, the correction phase will proceed undisturbed, and the NEAR will report failure. Thus the use of redundant operations, while restricting pipelining in this sense, will not destroy any information in the database. It should also be noted that, especially in large databases, the probability of a failed EXTRACT or NEAR will usually be very low.

The correctness of the algorithm can be demonstrated in two steps, which we outline here. First, note that a serial execution of the algorithm, where all of the steps corresponding to each operation are completed before the next operation is begun, has the usual radix tree semantics. Next, we observe that that pipelining preserves those semantics (except for the possibility of a failed operation, as above). In particular, the pipelined execution of a series of operations yields the same results as the serial execution of the same operations with the failed EXTRACTs and NEARs removed.

To see that pipelining works, note first that only operations which induce correction passes can cause a change in behavior, since every other operation proceeds in a strictly top-down fashion. Now we assert that all operations (except failed operations) have the same effect on the data structure and return the same results in the presence of correction passes as in the case where corrections are allowed to complete before other operations are begun. Failed operations have no effect at all.

Before examining each case, we note that the pipelined algorithm ensures that each node's count at any given time is at least as large as its "true" value, *i. e.* the value it would have if all operations in progress below it in the tree were completed (that is, all downward passes and correction passes finished). This is true because the completion of operations lower in the tree can only cause a count to be decremented, never incremented.

Now we examine the effect of correction passes on each operation type.

- FIND traces the tree based on the search key supplied as input. If the key has not been inserted or has been deleted, either part of that path in the tree will be missing or at least one correction pass will be waiting at the processor storing the leaf. When processing of the FIND reaches the leaf, exact information on the status of the string is available because all related previous operations have arrived already, and the appropriate answer is reported.

- A NEAR operation fails if and only if it enters upon a single-successor path containing a correction pass. Otherwise, it always returns the key nearest to the search key.

- An EXTRACT operation appropriately decrements counts, and fails in the same case as a NEAR operation. In this case it has no net effect on the data structure, since the decrementation it has performed has merely accelerated the effects of the correction pass causing it to fail.

- If an INSERT operation is redundant, it will always find counts of at least one on the path leading to the appropriate leaf, where complete information on the string's status is available. Thus the redundancy will be detected and a correction phase begun, as in the serial case. If an INSERT is not redundant, it will eventually either reach a juncture where part of the tree structure for the string is missing or reach a leaf representing the string where correction passes are waiting. In the first case nonredundancy is clear, and in the second case all of the information needed to make the decision is available at the leaf. In both cases, the appropriate modifications to insert the string in the tree are made.

- If a DELETE is redundant, it will reach either a missing part of tree structure or a leaf holding correction passes which will eliminate the string. In either case, the operation will have no effect. If a delete is nonredundant, it will eventually reach the leaf corresponding to the string to be deleted, where full information is available. In this case, a correction pass will start, just as in the serial case.

In summary: a radix machine uses L processors, each with memory capacity for at most N radix tree nodes. It accepts one operation per cycle, and has a latency of L cycles for all operations returning an answer. This constitutes linear speedup over the serial algorithm, and is as fast as possible given unit key bandwidth. Redundant operations can be performed, with the restriction mentioned above.

5.3.2. Tradeoffs and tuning

This basic algorithm admits a large number of modifications in the interest of memory savings or speed enhancements. Some of the variations on the serial algorithm can be used, such as the choice between implementing nodes as tables or linked lists, or the possibility of storing keys with their component symbols reversed. This section mentions a few other possibilities; many more are possible, especially given some knowledge of the statistics of the keys to be used.

A fundamental tradeoff is based on the size of the alphabet. If the keys are expressed in binary, more processors will be used than if the alphabet is larger, and more pointers will be used per key. On the other hand, this approach speeds the EXTRACT and NEAR operations, and will cut down on the average number of empty pointers in a node, possibly resulting in a net savings in hardware. In general, the radix can be tuned according to key statistics and implementation technology.

More tuning can be done by small specializations of the data structure. For example, special types of nodes can be used where there is only one descendant, saving on unused pointer space while keeping an efficient table structure for nodes with multiple descendants. In addition, only the first single-descendant node in a chain needs to keep a count, if interprocessor bandwidth is sufficient to carry a count downwards. An example of the power of the first technique is its application to large databases of English words, where it typically achieves a factor of four to five in memory compression (see Appendix 5.B).

Another opportunity for speed improvement is related to bandwidth between processors and their local memories, which will probably be the speed bottleneck for large databases. Memory transfers to and from the processor can be avoided by on-memory-board intelligence. In the case where nodes are implemented as linked lists, a list-searching unit might be added. Where nodes are implemented as tables, a "find first one-bit" device similar to a priority encoder could be used along with a small, specialized occupancy memory to quickly find the appropriate child in EXTRACT or NEAR searches.

5.4. Comparison of the different architectures

This section argues that, for most dictionary tasks of practical interest, the radix machines described above are more appropriate than tree machines. In many cases, they are also superior to the design of Carey and Thompson. The first part of this argument is given in terms of trends and rough comparisons of cost and performance. The second part consists of a description of hypothetical implementations of the radix machine and a tree machine. Carey and Thompson's scheme is omitted from the second comparison, as its difference in cost and performance from the radix machine is very sensitive to the assumptions made, and a specific comparison made at this level of detail cannot be very informative. An asymptotic counterpart of this comparison is given in Appendix 5.C.

5.4.1. Comparisons

The different possibilities for the implementation of dictionary operations can be compared in terms of hardware cost and execution speed. The radix machine and the Carey-Thompson machine have roughly similar hardware costs; they use a small number of processors and a large amount of memory. The radix machine may tend to use more memory, depending on node packing density, alphabet size, and key length; on the other hand, prefix sharing will tend to reduce memory requirements. Which system uses less hardware will depend on the characteristics of the keys to be stored.

On the other hand, the hardware cost comparison with tree achines boils down

to one between a 1) single processor and $O(N)$ nodes in memory, each holding some number of pointers, and 2) $\theta(N)$ small processors, each holding a few keys. Assuming a reasonable degree of complexity in a processor, and given current relative densities for memory and processing in, for example, MOS technology, there are very few cases where a radix machine is not significantly cheaper to build than a tree machine. The discussion below describes a pair of hypothetical implementations which support this point.

A comparison of the speeds of the different approaches hinges on the length of a key. If keys are small enough to be handled in a primitive operation of the machine in question, radix machines and Carey-Thompson machines should have roughly comparable performance. Tree machines may have a performance edge, since they will not need to access large memories and they have slightly simpler data structures. They will also have long wires, though (see Appendix 5.C), so interprocessor communication delays may dull that edge in large machines.

The picture changes, however, when keys are significantly longer than the native word size of the machine. While a radix machine approach can maintain constant throughput by using more processors for longer keys, the other types of machines must either widen their datapaths accordingly or simply serialize the communication and handling of keys.

The use of wide datapaths imposes penalties in both hardware complexity and cycle time. A key in a natural language database, for example, may have more than 100 bits, requiring expensive interprocessor connections and memory buses. Also, long keys must be compared in a single clock cycle; this comparison requires time at least proportional to the logarithm of the key length. This approach also has a negative impact on modularity and flexibility; tree machine chips must be designed to handle the largest key that can ever be handled, and components for a Carey-Thompson machine must either be maximum-sized or divide memory access and comparison arithmetic into slices, making it less convenient to do arithmetic in logarithmic rather than linear time.

The adoption of the serial approach, on the other hand, causes performance to

degrade in proportion to the length of the keys stored. This effect may, in fact, reduce the speedup that these approaches gain over a uniprocessor balanced tree algorithm. If the number of keys is small compared to the number of possible keys, a search will usually examine only a few symbols at each node until it reaches the bottom of the tree, since it will usually find a mismatch very quickly. Thus the time for a serial implementation to perform an operation may not increase in strict proportion to L. The tree machines and the Carey-Thompson machine, on the other hand, are limited by the need to transfer entire keys between processors and by the need to perform a full-length comparison in at least one node in the tree for each search. Because of this effect, it is even possible that a sparse set of keys could cause the Carey-Thompson and tree machines to run slower than a serial algorithm with a simpler inner loop.

Beyond the sheer cost of hardware and the long-key issue, the radix machine has several other advantages, which are shared by the Carey-Thompson machine. First, off-the-shelf memory chips and bit-slice processor components can be used instead of custom chips. Besides avoiding the cost of custom LSI design, this allows the use of technologies which have been highly optimized for different goals: the memory technology for density, and the processor technology for speed. Custom tree processors are unlikely to be simultaneously dense and fast.

Second, because each processor can be more expensive than when a million of them are used, processors can be much more flexible and powerful. Most of the tradeoffs and performance tunings mentioned in the previous section can be made without hardware modification. The dictionary operations can be modified and subtle variations introduced. None of this is feasible with a tree processor whose control circuitry needs to be as simple as possible.

Finally, a radix machine is not as "single-purpose" as a binary tree dictionary machine. In particular, the parallel radix tree algorithm fits well on a general-purpose linearly connected multiprocessor. Such a machine could be extremely useful in the performance of many regular-structured computations, especially systolic algorithms. Conversely, given any general-purpose, tightly coupled multiprocessor architecture, it would be fairly inexpensive to add the linear connections

needed to allow the efficient implementation of the radix machine algorithm. Since a number of memory accesses must be made for each logical cycle, these links would only need to run at some fraction of local memory speed.

One problem to which the radix machine and, to a lesser extent, Carey and Thompson's machine are subject is memory allocation between processors. Since the number of keys stored does not uniquely determine how much memory is needed at which level of the tree, extra space will usually need to be allocated. This will often not present a problem in practice, as the statistics of large databases will usually tend to vary slowly, and memory allocation can be done infrequently at the board level.

5.4.2. Hypothetical implementations

This section compares radix and tree machines for dictionary operations on databases of about one million English words. It is reasonable to suppose that databases much larger or much smaller than this (*i. e.*, by a factor of 100 or more) are not likely to be well suited to these approaches: collections of a few thousand keys can be quickly accessed by a uniprocessor, and collections of many millions of keys are likely to be kept in secondary storage.

The comparison uses the following assumptions, which are somewhat biased in favor of binary tree machines:

- Keys are represented as strings of six-bit bytes. This is well-suited to alphabetic data, and will help the tree machine to be decomposed onto low-pinout chips. The maximum key length allowed is 16 bytes.

- The tree machine considered is Somani and Agarwal's, which requires the smallest amount of internode I/O among the pure binary tree machines and wastes the fewest processors among the machines performing redundant operations. Since COMPRESS operations need not, in fact, be done in every cycle, they are not counted here. If Somani and Agarwal's scheme were followed exactly, the implementation described here would run twice as slow.

- A tree processor (containing key registers, three bidirectional ports, an ALU, and a relatively complex controller) occupies one mm^2 of chip area. A single chip, organized according to a tree decomposition scheme proposed by Leiserson [33], holds 32 processors and four off-chip ports

and fits in a 28-pin DIP. (This density is several years from feasibility, and it is questionable whether the four pins left will suffice for power, synchronization, and control.)

- Circuit boards for both machines hold 100 28-pin DIPS.
- The radix machine is built using 256K-bit memory chips. Memory boards use 25 chips for peripheral circuitry. A memory board thus holds $75 \times 256K \div 6$ bytes, more than 3 MB. For simplicity, 3 MB is assumed.
- A processor board holds a processor and .5 MB of memory.
- The memory cycle time for the radix machine is 500 ns.
- Cycle time for the tree machine, dominated by the time to drive signals between boards, is 250ns.

Given these assumptions, the binary tree machine needs about 30,000 chips and hence 300 boards. Since each key step potentially requires transmitting a key to and from the parent node, at least 32 machine cycles, or 8 μs, are needed per operation. Based on statistics gathered from several large dictionaries of English words (see Appendix 5.B), the radix machine (using a special representation for nodes with only one successor) would need 16 processor boards and 26 extra memory boards, for a total of 42. Especially since 25 chips on the memory boards is enough to build in some intelligence, it is reasonable to assume that one logical cycle can be carried out within 16 memory cycles, or 8 μs, thus achieving the same performance as the tree machine with about 14% of the hardware cost. Appendix 5.B further indicates that a tree machine with *any* set of one million 16-byte keys will require less than 40% as much hardware as a million-node tree machine.

5.5. Conclusions

The discussions above show that, in most practical cases, a radix machine with a small number of processors is preferable to a tree machine for rapid execution of dictionary tasks. In addition to lower hardware cost and higher speed for longer keys, the radix machine has practical advantages of implementability, flexibility, and generality. Where keys are relatively short and their statistics cause poor utilization of a radix machine's memory, Carey and Thompson's design may be preferable.

Tree machines can be expected to be of more value in applications where a query or update requires $O_k(n)$ time on a uniprocessor.

More generally, this example serves to show that VLSI architectures with large numbers of processors are not always the best approach to constructing high-performance systems. Especially where asymptotic performance gains are small and the asymptotic growth in processor count is large, architectures with fewer processors and more memory may be preferable.

5.A. Tree machines

This appendix describes the tree machines discussed briefly in Section 5.2. It begins by describing a very simple linear pipeline architecture that can perform the dictionary operations, and then treats each of the tree architectures in turn.

A very straightforward architecture implementing some of the dictionary operations is Kung and Leiserson's systolic priority queue [32], which uses a linear array of N processors, each storing one key, to perform INSERT and EXTRACTMIN operations. Each of these operations is performed with $O_k(1)$ period and latency. Leiserson notes that this scheme can be extended (by folding the array in half) to include the EXTRACTMAX operation, maintaining the same performance. In fact, the DELETE, FIND, and NEAR operations can also be added, although the latter two have $O_k(n)$ response time. Insertions and deletions may be redundant.

In the same paper, Leiserson also gives a binary tree architecture (again, designed only as a priority queue) that can support all of the dictionary operations, barring redundancy, with $O_k(1)$ pipeline period. The EXTRACTMIN operation has latency $O_k(1)$, and the other value-returning operations have latency $O_k(\log N)$. If a DELETE is redundant, a valid key can be destroyed; if an INSERT is redundant, a hole in the data structure is created which wastes a processor. It might be noted that the folding trick mentioned above can bring the latency for EXTRACTMAX down to $O_k(1)$.

Ottman *et al.* [40] give a tree machine design whose topology is actually an X-tree, where all of the nodes on one level of the tree are strung together in a row. This

feature makes implementation less convenient. Their design cuts Leiserson's $\log V$ latencies down to $\log n$. This design also supports redundant insertions and deletions with the use of COMPRESS operations which eliminate holes in the data structure. The cost of this flexibility is that $O(n)$ holes may exist at any time, wasting that many processors.

Atallah and Kosaraju [2] describe an improved design that has all of the properties of the previous one except that it supports EXTRACTMIN with $O_k(1)$ latency and has a pure binary tree structure. This design still suffers from the wasted processor problem.

Most recently, Somani and Agarwal [45] have proposed a tree-structured machine in which the keys are unordered, which simplifies the algorithm in some respects. It also allows a scheme for handling redundant operations in which only $\log n$ processors can be wasted. Their method of keeping the tree balanced, however, does not lend itself to modular implementation. Inserted keys are added at the bottom of the tree across its breadth, rather than on one side, as in some of the other machines. As a result, the machine can be conveniently enlarged only by factors of two. The other machines, if decomposed onto chips according to Leiserson's scheme [33], may have any number of processors in the bottom level of the tree which is a multiple of the number of leaf cells on a chip.

Table 5-1 summarizes the latencies and some other properties of these machines. XMIN and XMAX abbreviate EXTRACTMIN and EXTRACTMAX, and the machines are listed by first author.

5.B. A radix machine size estimate

This appendix estimates the memory requirements for a radix machine, as described in Section 5.4, containing one million keys of up to sixteen six-bit bytes. An estimate is made based on a large collection of English words, and a crude worst case giving an absolute upper bound is also given.

The English language statistics used here are drawn from an online copy of

Table 5-1: Tree machine latencies.

machine	FIND	XMIN	XMAX	*comments*
Leiserson	log V	1	log V	redundant INSERT/DELETE corrupts database.
Ottmann	log n	1	log n	redundant operations OK. up to n wasted processors. X-tree topology.
Atallah	log n	1	1	redundant operations OK. up to n wasted processors.
Somani	log n	log n	log n	redundant operations OK. up to log n wasted processors. not modular.

Webster's Second International dictionary, containing 235,405 words. Several smaller wordlists were also consulted, with similar results. Memory requirements for one million words were extrapolated from dictionary measurements by scaling by a factor of 4.5, thereby adding a 5% margin of extra capacity.

Radix tree nodes are represented as tables with twenty-six entries, with one specialization. Nodes with fewer than two children are coded in nine bytes: one for a character, four for an address, and four for a count. The space required for these nodes could be reduced further by eliminating all but the first count in a string of such nodes (most nodes would then require just four bytes), but that option is not taken here to avoid the proliferation of node types. Nodes with more than one child are allocated 108 bytes, four for each descendant address and four for a count.

Of about 3.4 million nodes required, about 87% have no more than one child. This allows a reduction in node space by a factor of slightly less than five. The number of boards required per processor is as follows, for a total of 42:

P:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#:	1	1	2	3	4	5	5	4	4	3	2	2	2	2	1	1

It is also interesting to consider the worst-case memory requirements, where every word is sixteen characters long and nodes are used as inefficiently as possible. An

exact calculation of the worst case is difficult, since the tradeoff between sharing and node efficiency is complex. As a crude upper bound, however, note that at most one million large nodes can be used to produce one million leaves. Furthermore, at most 16 million small nodes can be used, one per key per processor. The small nodes account for 3 memory boards per processor. The large nodes require 108 Mbytes, or 36 boards' worth; disregarding memory on processor boards and assuming the worst possible distribution of nodes over processors, at most 16 extra boards are needed, for a total of 52. Taking this figure, adding both (incompatible) worst cases together, and including processor boards yields a total board count of 116, still significantly fewer than needed for a tree machine.

5.C. A theoretical comparison of dictionary machines

This section compares the asymptotic time and hardware complexities of the two types of schemes, and gives some observations on how these results should be interpreted. The main difficulty with asymptotic measures in this case is that many of the interesting comparisons to be made hinge on factors of $\log N$, which cannot in practice be very large; in fact, the "constant factors" which are disregarded in such analyses are often of equal importance. Nonetheless, an asymptotic point of view provides a framework within which very different structures can be compared.

In assessing time and hardware requirements, we will in fact discuss three quantities: hardware area, pipeline period in basic machine cycles, and the amount of time needed to perform a machine cycle. The product of the latter two quantities gives the actual period. Machine cycle time will depend on maximum wire lengths in the machine, which are in turn derived from the hardware area.

One extra complication arises in comparing quantities of $O_k()$ and $O_s()$. In what follows, L, the bit length of the longest possible key, is taken to be $\theta(\log N)$.

Hardware area A binary tree machine has N processors, each of which stores $L = \theta(\log N)$ bits of data. Hence its total area is $\Omega(N \log N)$. A radix machine has $L = \theta(\log N)$ processors, each with a memory holding, in the worst case, N nodes containing $\log N$-bit pointers. The total hardware complexity of this machine, then, is $O(N \log^2 N)$. In practice, the factors of $\log N$ are of questionable import, since

the major comparison to make is between one processor per word and some small multiple of $\log N$ bits of memory per word.

Machine cycles A pipeline period for the radix machine takes $O(1)$ cycles, independent of the size of the keys or of the size of the database. The pipeline period for a tree machine depends on whether keys are handled serially or in parallel. If they are handled serially, $L = \theta(\log N)$ cycles are needed. If they are handled in parallel only $O(1)$ cycles are necessary, but this will typically be a very costly option.

Cycle time Following the method of Paterson, *et al.* [41], it is easy to show that the smallest possible maximum wire length for a binary tree machine is $\Omega(\sqrt{N \log N} / \log N)$, or $\Omega(\sqrt{N / \log N})$. The longest wire in a radix machine processor memory will be on the order of the diameter of a single memory array, or $O(\sqrt{N \log N})$.

The question of how to derive delays from wire lengths is complex. A simple asymptotic lower bound, $delay = \Omega(wirelength)$, is given by the speed of light. Depending on conditions, models ranging from $\theta(1)$ to $\theta(wirelength^2)$ may be appropriate [37]. In practical cases, where most of the wire length is off-chip and systems are physically small enough that other effects dominate the speed of light, delays are less than linear in wire length. Thus the $\log N$ difference in longest wire length in this case is probably insignificant.

Results The two schemes are roughly comparable in asymptotic complexity, with the tree machines having a $\log N$ advantage in area and wire length, and the radix machine having a $\log N$ advantage in cycles needed for a pipeline stage over a serialized tree machine. As the example in Section 5.4.2 shows, the two complexity factors that are most telling in practice are the (constant) difference in cost between a processor and a small amount of memory, and the time cost of having a byte-serial machine.

6 Conclusions

Advances in electronics fabrication technology have made possible the design and construction of architectures that were previously unthinkable. This dramatic expansion of the computer design space calls for new designs and new methods of design and analysis. The prospect of machines with more than a "small constant number" of elements opens the door to measures of cost and performance at a higher level of abstraction than chip counts and clock rates. This thesis makes a number of contributions in this context.

Programmability Chapter 2 outlines the case for programmable architectures for systolic algorithms. It then describes the PSC, a concrete example of an architecture that is useful for a broad class of systolic algorithms. The PSC is architecturally far more efficient for the implementation of systolic algorithms than are ordinary microprocessors, and has been successfully fabricated, tested, and demonstrated. Section 2.2 then discusses alternative organizations for programmable implementations, which are suitable under different circumstances. This section clarifies issues in the use of local memory involving cost, performance, and modularity; it also presents a novel means of separating computation from data flow which allows modular systolic implementations using non-VLSI parts.

The notion of a general-purpose part intended for use in special-purpose architectures is a powerful one. First, it relieves the special-purpose system implementor of the burden of designing hardware from scratch. Second, it frees the individual processors from the necessity of efficiently supporting the full range of operations required by general-purpose computing, allowing more resources to be devoted to communication and functional parallelism. The on-chip horizontal microcode of

the PSC is a good example of this tradeoff; a fully general processor would need to be able to run large programs, necessitating an off-chip, hence narrower, path to code memory.

As noted in Chapter 2, a PSC-like architecture can be viewed as a smart arithmetic unit, rather than as a simple processor. Given some careful interface engineering, lacking in the PSC design, such a component could be extremely useful for the implementation of a large class of systems. A related approach is taken by the INMOS "transputer" [22], a microprocessor designed especially to be used in parallel architectures. The difference is that the transputer is intended for use in general-purpose multiprocessors, as well as in special-purpose systems; thus it cannot take advantage of the tradeoff mentioned above to provide highly cost-effective small-grain parallelism.

Synchronization Chapter 3 begins by establishing, apparently for the first time, a simple and intuitive mathematical model of clock speed and its limiting factors. It improves on some earlier work by clarifying the distinction between clock skew and clock delay. It suggests, also apparently for the first time, the use of on-chip pipelined clocking to avoid the problem of clock delay, and suggests conditions where this technique may be applicable. It proposes two variant models of clock skew, corresponding to differing conditions which may arise in practice. In view of these models, it then provides upper and lower bounds of clock skew on arrays of varying topologies.

The question of whether pipelined clocking on chips or wafers is a practical solution to clock distribution problems awaits experimental studies for resolution. In any case, the results of Sections 3.4 and 3.5 apply to any pipelined clocking scheme, on or off chips.

Self-timing has the attractive feature for local communication architectures that synchronization, too, is localized. The hybrid scheme discussed in Section 3.6, by encapsulating the self-timed elements of the design into a synchronization subsystem, offers a promising means of taming the complexity of implementing such systems.

Serialized implementations Chapter 4 discusses two aspects of the use of bit-serial and other serial approaches to systolic design. First, it shows how serial designs may be derived from parallel designs. It then examines some of the practical cost/performance issues that arise in such designs, and presents a list of factors which limit the effective use of highly serialized schemes.

Dictionary machines and practical issues Chapter 5 examines, as a case study, the subject of machines for dictionary operations. It points out a number of ways in which several previous efforts have been unrealistic in their modeling of the problem, most notably in the implicit assumption that keys in the dictionary have constant length, independent of the number of entries. It describes a new architecture and algorithm that take this into account, and have a number of other practical advantages over earlier proposals. Finally, it considers a set of practical issues that are usually ignored in paper architectural studies, but can have a pivotal effect on the choice of one architecture over another.

In addition to the dictionary problem itself, this study also touches on two issues of wider import. One is the tradeoff found between processors and memory capacity, which is a twist on the standard tradeoff between time and space. The second is the question of appropriate measures of cost and performance for special-purpose architectures.

The processor-memory tradeoff is one that arises in several cases where a systolic array with $O(N)$ processors takes $O(N)$ time, for a total operation count of $O(N^2)$, and a $O(N \log N)$ serial algorithm exists. Related to the dictionary problem are such cases as priority queues and sorting, where $O(\log N)$ processors can do the job in time $O(N)$. Another case is digital filtering, which can be done serially with fast Fourier transforms in time $O(N \log N)$. Here too, $O(\log N)$ processors with $O(N)$ memory apiece can achieve the same asymptotic performance as the straightforward algorithm on $O(N)$ processors. It would be interesting to find examples of this phenomenon for problems of higher complexity. It should be noted that memory-intensive architectures are not always preferable to processor-intensive structures. The practical evaluation of which approach is better will depend on constant factors and on the size of the real-world data encountered.

Constant factors also impinge heavily on the second issue mentioned above, that of complexity measures for special-purpose architectures. It is tempting to think of such architectures as a particular kind of algorithm, and to apply the usual techniques of analysis. This approach falls short in two ways. First, asymptotic analysis is often of little practical value for analyzing architectures, simply because physical machines tend to be small. Files with hundreds of thousands of records are common; machines with hundreds of thousands of processors are not. Second, algorithms are usually described with reference to a standard RAM model of computation: counting operations or program statements will usually lead to a realistic estimate of runtime, and space estimates are equally available. This is not true of special-purpose architectures, where differences in suitable technologies or the potential for inexpensive dedicated hardware can play a critical role. It is an important open question whether such factors can be captured in an abstraction which is at once both general and useful.

Appendix A
PSC reference manual

This manual is intended as a reference for the microcoding of the PSC, and does not deal with its silicon implementation or with details of interfacing. The overall structure is described, and the details of the operation of each unit in the processor are described individually. Microcode mnemonics are displayed in a `typewriter font`. Bits in all fields are numbered from the right, beginning with zero.

A.1. Overall structure

The PSC is designed as a connection of function blocks. These blocks are divided into a data part and a control part. This section briefly describes their function during normal operation (that is, exclusive of microcode loading and refreshing.)

The function blocks in the data part operate in a two-phase regime: during phase 1 (φ_1) each block computes new values for its output registers and internal state from its internal state and the contents of its input registers, and during phase 2 (φ_2) information is passed from output registers to input registers, and internal state is updated. Input registers are transparent during φ_2, and output registers are transparent during φ_1.

The control part operates so that a particular microinstruction persists from a phase 2 through the following phase 1; thus an instruction determines the inputs of each block and the operations to be performed on those inputs. This is done by reading out a new instruction just before φ_2 begins. During φ_2, the sequencer uses instruction bits and condition bits computed during the preceding φ_1 to compute a new address; thus branching is immediate, but depends on the condition bits of the preceding instruction.

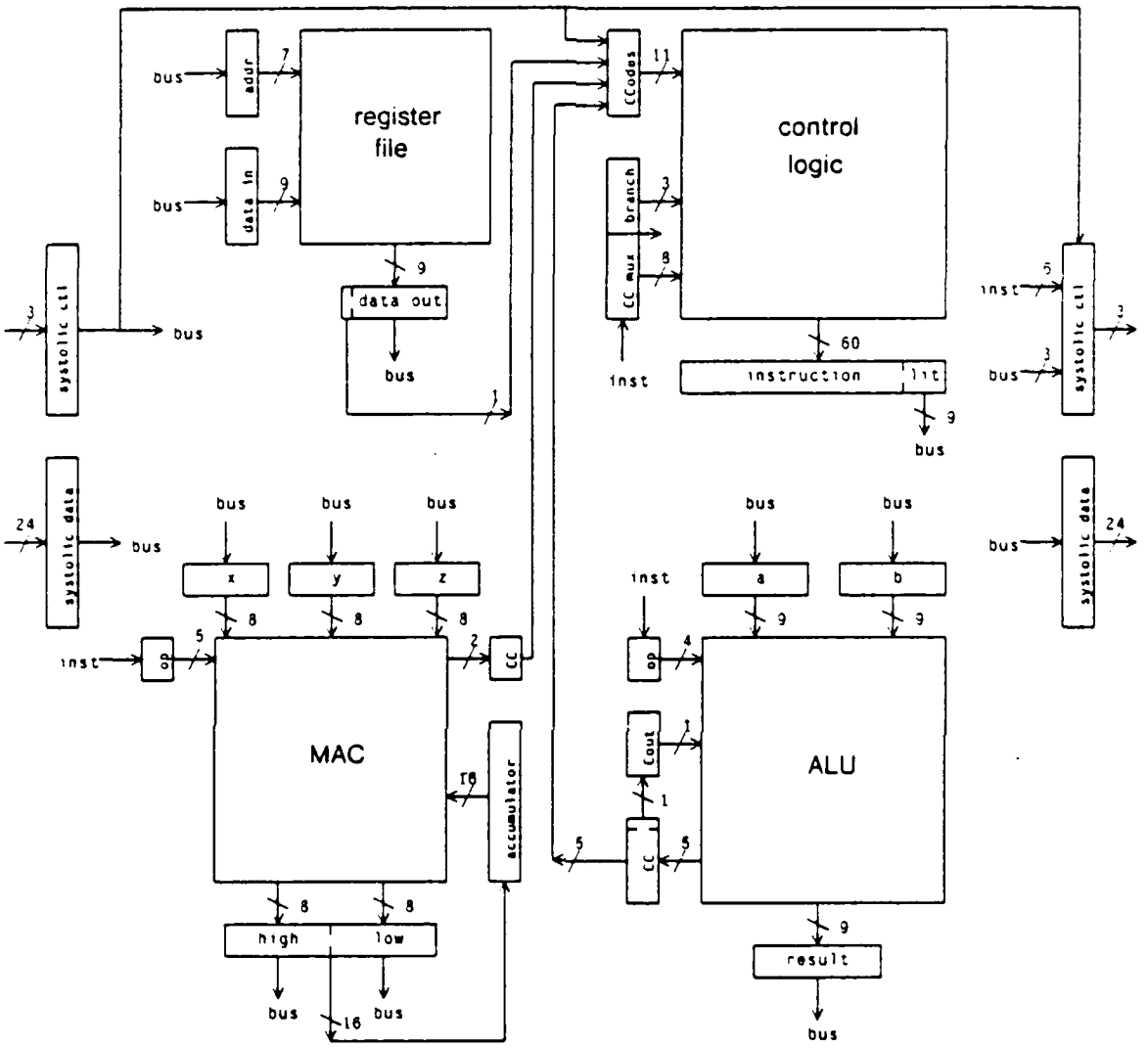


Figure A-1: PSC blocks.

The main function blocks, as shown in Figure A-1, are:

- A 64 word × 60 bit control store and its addressing logic. Inputs are condition codes from the rest of the chip, as well as branch control and condition code select fields from the most recent microinstruction. The sole output is the microinstruction register. Internal state and logic include a program counter, subroutine return address stack, and the control store itself.

- An eight (+ a little) bit ALU. The operation to be performed is determined by a four bit opcode. Input registers hold two nine bit operands,

along with the carry out from the previous ALU operation. Output registers hold a nine bit result and five condition code bits (negative, zero, positive, carry, and overflow).

- An eight bit multiplier-accumulator (MAC). The mac is controlled by five instruction bits. Input registers hold three eight bit operands, along with the 16 bit result from the previous cycle and a single carry/overflow bit. Output registers hold the high and low eight bits of the result, along with two condition codes bits (overflow and leftmost bit).

- A 64 × nine register file. The register file is controlled by a microinstruction read/write bit. Input registers hold a six bit address and a nine bit data input. The single output register is a nine bit data output. Internal state consists simply of the contents of the registers. Although the timing of the register file is actually the same as that of the microcode store, it behaves in accordance with the simple two-phase scheme.

- Three eight bit systolic data and three one bit systolic control inputs. These inputs to the chip contain only output registers, whose contents come from off-chip. Note that inter-cell communication thus takes place during phase 1, the computation phase.

- Three eight bit systolic data and three one bit systolic control outputs. The data outputs contain only input registers, whose contents are delivered off-chip. The control outputs contain some selection logic as well as input registers.

In each cycle, the exact connectivity of input registers to output registers, as well as the decision of input registers between accepting new values and holding their previous values, is controlled by the current microinstruction. In particular, the ALU operands, the three eight bit MAC operands, the register file's address and data inputs, and the systolic outputs can either hold their previous values or obtain new values from one of three buses. The buses, in turn, obtain their values from among the ALU output, the MAC high and low outputs, the register file output, a nine bit literal field in the microinstruction, and the systolic inputs.

A.2. Microinstruction format

The microinstruction contains the following fields:

- Bus source addresses - 9 bits (3 buses × 3 bits for 8 output registers)
- Input register control - 20 bits (10 registers × 2 bits for 3 buses plus hold)
- Systolic control output - 9 bits (3 bits of control per systolic control output)
- ALU operation - 4 bits
- MAC operation - 5 bits
- Register file control - 1 bit
- Program branching control - 3 bits
- Literal or condition code specifier - 9 bits

Thus, a microinstruction requires a total of 60 bits.

A.3. Bus connectivity

Each of three system-wide buses carries a nine bit value, and may be read by any of the ten input registers which read the buses. The output registers read by the buses (`Bus1`, `Bus2`, `Bus3`) are selected by the bus source addresses as follows:

0. ALU result (`ALU`)
1. MAC high-order eight bits (right adjusted with zero fill) (`Hi`)
2. MAC low-order eight bits (right adjusted with zero fill) (`Lo`)
3. register file output (`MDO`)
4. systolic data A extended on the left by systolic control A (`SDA`)
5. systolic data B extended on the left by systolic control B (`SDB`)
6. systolic data C extended on the left by systolic control C (`SDC`)
7. microinstruction literal field (`LitVal`). *caveat programmitor:* `LitVal` *occupies the same bits as the condition code specifiers (*`CC1` *and* `CC0`*); it should not appear in the same microcode source instruction with them.*

Each of the input registers which can read the buses holds its current value if its

control bits are zero (Hold), or selects bus 1, 2, or 3 (Val1, Val2, Val3) according to the numeric value of the control bits.

A.4. Control logic

The control logic function block consists of a current address register, next address logic, the control store, and a four × six bit pushdown stack for return addresses. During phase 2, the logic in the block computes a new instruction address from the current address register, condition code bits, and fields from the current microinstruction, and also computes new stack contents. During phase 1, the new current address is used to update the current address register, and the stack storage is likewise updated. The next instruction to be executed is selected from the the control store array at this time. This structure is sketched in Figure A-2, in which the phases during which each register is updated are indicated.

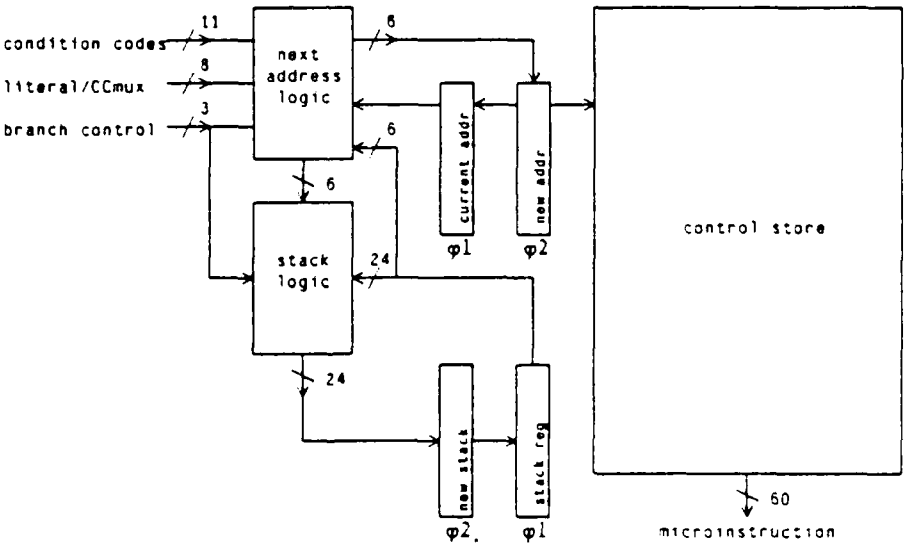


Figure A-2: Control logic.

The next address logic contains two condition code multiplexers, each of which chooses one of the chip's condition bits. The two bits are concatenated to form a two bit number which may be used in the computation of the next instruction address. The multiplexers are controlled by two four bit fields, CC1 and CC0, which

occupy the low-order eight bits of the microinstruction literal field. The condition code bits are selected as follows:

0. constant zero (`Zero`)
1. constant one (`One`)
2. systolic control A (`SCA`)
3. ALU negative (`AluN`)
4. ALU positive (`AluP`)
5. ALU zero (`AluZ`)
6. ALU overflow (`AluV`)
7. ALU carry out (`AluC`)
8. systolic control B (`SCB`)
9. MAC leftmost bit (`MacL`)
10. MAC overflow (`MacV`)
11. systolic control C (`SCC`)
12. leftmost bit of register file output (`RegBit8`)

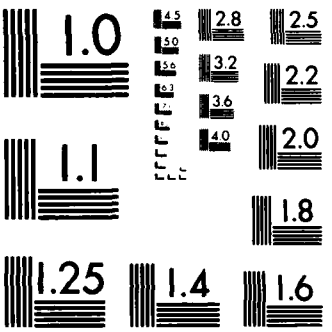
The next address and new stack contents are computed, depending on the three bit program branch control field `Jump`, as follows (where all arithmetic is modulo 64):

0. (`Seq`) next address = current address + 1.
1. (`Call`) next address = contents of literal field, push current address + 1 onto stack.
2. (`Return`) next address = contents of top of stack + two CC mux bits, pop stack (the obvious bounded stack considerations apply).
3. (`ToLit`) next address = contents of literal field.
4. (`OnCC1`) next address = current address + 1 + two CC mux bits. This allows branching to one of four different addresses, depending on two bits.
5. (`OnCC0`) next address = current address + two CC mux bits. This allows single-instruction "wait" loops.

Note that the condition code specifiers share their space in the microinstruction with the literal field.

AD-A156 699 IMPLEMENTATION ISSUES FOR ALGORITHMIC VLSI (VERY LARGE SCALE INTEGRATION)..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE A L FISHER OCT 84 2/2

UNCLASSIFIED F33615-81-K-1539 F/G 9/5 NL



A.5. ALU operation

The ALU, with one exception, generally uses the eight low-order bits of its inputs *a* and *b* to compute a nine bit result consisting of an eight bit logical or arithmetic result extended on the left with a carry out. The exception is the pass instruction, which simply sets the nine bit output to the nine bit input *a*. Condition code bits are defined as follows:

- negative - left bit of eight bit result
- zero - one iff all eight bits of the result are zero
- positive - one iff neither negative nor zero are true
- carry - carry out of the left end of an eight bit operation
- overflow - the exclusive or of the carries into and out of the left end of an eight bit operation.

The ALU operations (`AluOp`) are as follows. Note that operations 0 through 3 always generate carries of zero, and that arithmetic operations are in twos complement unless otherwise noted.

0. (`And`) *a* AND *b*.
1. (`Xor`) *a* XOR *b*.
2. (`Or`) *a* OR *b*.
3. (`Not`) NOT *a*.
4. (`PassA`) *a* as a nine bit value.
5. (`Add`) *a* + *b*.
6. (`CAdd`) *a* + *b* + previous carry out. Used in multiple precision addition.
7. (`Inc`) *a* + 1.
8. (`CInc`) *a* + previous carry out.
9. (`Sub`) *a* − *b*.
10. (`CSub`) *a* − *b* + previous carry out − 1. Used in multiple precision subtraction.
11. (`Dec`) *a* − 1.
12. (`CDec`) *a* − previous carry out.

13. (Neg) $0-a$.
14. (Cmp) $a-b$ unsigned. Used to compare characters and in finite field operations -- carry out indicates $a<b$, zero CC indicates $a=b$.
15. (Shl) a shifted left one position (rightmost bit = 0, leftmost bit shifted into carry.)

A.6. MAC operation

The multiplier has the capability of multiplying two eight bit numbers, x and y, each of which may be independently signed or unsigned, and adding in an eight bit number z or the contents of a sixteen bit accumulator. The multiplier's output is signed if either x or y is signed. The MAC has five control bits, which operate as follows:

- X signed -- if 0 (MacXu), treat x input (MacX) as an unsigned number; if 1 (MacXs), treat x as a twos complement number.
- Y signed -- similarly for the y input.
- Carry in (MacCin) -- used as described below, for multiple precision arithmetic.
- Op code (MacOp) -- two bits:

    0. (AddZu) Multiply x and y and add eight bit z (MacZ) input as an unsigned number. Add Carry in $\times 2^8$.
    1. (AddZs) Multiply and add z as a signed number. If the previous result was negative, add -2^8. This is used for multiple precision operations.
    2. (NoAcc) Multiply x and y. Add Carry in $\times 2^8$.
    3. (Acc) Multiply and add accumulator (last result) value. Add Carry in $\times 2^8$.

The multiplier also produces two condition codes: overflow (MacV), signifying that the result exceeds sixteen bits, and leftbit (MacL), which represents the leftmost bit of a seventeen bit result. Note that these can differ only in the signed case.

A.7. Register file

If its read/write control bit is zero (`Read`), the register file sets its output register to the value addressed by its address input register. If the bit is one (`Write`), it writes the value in its data input register into the location addressed by its address input register. Note that this implies that for a read operation, an address must be supplied in the cycle before the data is used.

A.8. Systolic outputs and inputs

The systolic data outputs (`SDAout`, `SDBout` and `SDCout`) are viewed as input registers in the abstract function block scheme, and operate in the same way as other input registers that read the buses. The systolic control outputs (`SCAout`, `SCBout` and `SCCout`) are controlled by three bit control fields as follows:

0. (`Hold`) hold.
1. (`Bus`) leftmost bit of the bus read by the corresponding systolic data output (held if the data output is holding).
2. (`Zero`) constant zero.
3. (`One`) constant one.
4. (`Pass`) current value of the corresponding systolic control input (e. g., SCA.Out = SCA.In).

The chip's systolic inputs, which are viewed as output registers in the abstract scheme described above, get their values from neighboring chips or a host during the computation phase (phase 1). Thus a result which is computed in a given chip can be computed on by a neighboring chip not in the *next* computation phase, but in the one after that. The sequence is as follows:

φ_1: compute on chip 1.

φ_2: send results over buses to output port on chip 1.

φ_1: send values from output port of chip 1 to input port of chip 2.

φ_2: send values from input port of chip 2 to functional unit.

φ_1: compute on chip 2.

References

1] R. Aleliunas and A. L. Rosenberg.
On embedding rectangular grids in square grids.
IEEE Transactions on Computers C-31(9):907-913, September, 1982.

2] Mikhail J. Atallah and S. Rao Kosaraju.
A generalized dictionary machine for VLSI.
Technical Report JHU 81-17, Johns Hopkins University, Department of Electrical Engineering and Computer Science, 1981.

3] A. J. Atrubin.
A one-dimensional real-time iterative multiplier.
IEEE Transactions on Electronic Computers 14(3):394-399, June, 1965.

4] M. R. Barbacci.
Instruction set processor specifications (ISPS): the notation and its application.
IEEE Transactions on Computers C-30(1):24-40, January, 1981.

5] K. E. Batcher.
Design of a massively parallel processor.
IEEE Transactions on Computers :836-840, September, 1980.

6] J. L. Bentley and H. T. Kung.
A tree machine for searching problems.
In *Proceedings of the 1979 International Conference on Parallel Processing*, pages 257-266. IEEE, August, 1979.

7] J. Blackmer, P. Kuekes and G. Frank.
A 200 MOPS systolic processor.
In *Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV*. The Society of Photo-Optical Instrumentation Engineers, August, 1981.

8] R. P. Brent and H. T. Kung.
Systolic VLSI arrays for linear-time GCD computation.
In F. Anceau and E. J. Aas (editors), *VLSI 83*, pages 145-154. North-Holland, August, 1983.
A revised version is to appear in *IEEE Transactions on Computers*.

9] K. Bromley, J. J. Symanski, J. M. Speiser and H. J. Whitehouse.
Systolic array processor developments.
In H. T. Kung, R. F. Sproull and G. L. Steele, Jr. (editors), *VLSI Systems and Computations*, pages 273-284. Carnegie-Mellon University, Computer Science Department, Computer Science Press, Inc., October, 1981.

10] Michael J. Carey and Clark D. Thompson.
An efficient implementation of search trees on $\lceil \lg N+1 \rceil$ processors.
IEEE Transactions on Computers C-33(11):1038-1041, November, 1984.

11] D. Cohen and G. Lewicki.
MOSIS -- The ARPA Silicon Broker.
In *Proceedings of the Second Caltech Conference on VLSI*. California Institute of Technology, January, 1981.

12] Alan Corry and Kaushik Patel.
A bit-sliced CMOS correlator.
In *Proceedings of 1983 International Symposium on VLSI Technology, Systems and Applications*, pages 134-137. 1983.

13] R. A. Evans, D. Wood, K. Wood, J. V. McCanny, J. G. McWhirter and A. P. H. McCabe.
A CMOS implementation of a systolic multi-bit convolver chip.
In F. Anceau and E. J. Aas (editors), *VLSI 83*, pages 227-235. North-Holland, 1983.

14] Allan L. Fisher and H. T. Kung.
Special-purpose VLSI architectures: general discussions and a case study.
In S. Y. Kung, H. J. Whitehouse and T. Kailath (editors), *VLSI and Modern Signal Processing*. Prentice-Hall, 1984.

15] Allan L. Fisher, H. T. Kung, Louis M. Monier and Yasunori Dohi.
Architecture of the PSC: a programmable systolic chip.
In *Proceedings of the Tenth International Symposium on Computer Architecture*. June, 1983.

16] Allan L. Fisher, H. T. Kung, Louis M. Monier, Hank Walker and Yasunori Dohi.
Design of the PSC: a programmable systolic chip.
In R. Bryant (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 287-302. California Institute of Technology, Computer Science Press, Inc., March, 1983.

17] Allan L. Fisher, H. T. Kung and Kenneth Sarocky.
Experiments with the CMU programmable systolic chip.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*. The Society of Photo-Optical Instrumentation Engineers, August, 1984.

18] M. J. Foster and H. T. Kung.
The design of special-purpose VLSI chips.
IEEE Computer 13(1):26-40, January, 1980.

19] Mark A. Franklin and Donald F. Wann.
Asynchronous and clocked control structures for VLSI based interconnection networks.
In *Proceedings of the Ninth Annual Symposium on Computer Architecture*, pages 50-59. April, 1982.

20] L. J. Guibas, H. T. Kung and C. D. Thompson.
Direct VLSI implementation of combinatorial algorithms.
In *Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 509-525. California Institute of Technology, January, 1979.

21] Frederick C. Hennie III.
Iterative Arrays of Logical Circuits.
M.I.T. Press, 1961.

22] INMOS Corporation.
IMS T424 transputer / advance information.
Product brochure, 1984.

23] Donald E. Knuth.
The Art of Computer Programming. Volume 3: *Sorting and Searching.*
Addison-Wesley, 1973.

24] H. T. Kung.
Special-purpose devices for signal and image processing: an opportunity in VLSI.
In *Proceedings of the SPIE, Vol. 241, Real-Time Signal Processing III*, pages 76-84. The Society of Photo-Optical Instrumentation Engineers, July, 1980.

25] H. T. Kung.
Use of VLSI in algebraic computation: some suggestions.
In P. S. Wang (editor), *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 218-222. ACM SIGSAM, August, 1981.

26] H. T. Kung.
Why systolic architectures?
IEEE Computer 15(1):37-46, January, 1982.

27] H. T. Kung.
Systolic algorithms for the CMU Warp processor.
In *Proceedings of the Seventh International Conference on Pattern Recognition*, pages 570-577. July, 1984.

28] H. T. Kung and M. Lam.
Wafer-scale integration and two-level pipelined implementations of systolic arrays.
Journal of Parallel and Distributed Computing 1:32-63, 1984.
A preliminary version appeared in *Proceedings of the Conference on Advanced Research in VLSI*, MIT, January 1984.

29] H. T. Kung and C. E. Leiserson.
Systolic arrays (for VLSI).
In I. S. Duff and G. W. Stewart (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.
A slightly different version appears as Section 8.3 of Mead and Conway [37].

30] H. T. Kung and S. W. Song.
A systolic 2-D convolution chip.
In K. Preston, Jr. and L. Uhr (editors), *Multicomputers and Image Processing: Algorithms and Programs*, pages 373-384. Academic Press, 1982.

31] S. Y. Kung and R. J. Gal-Ezer.
Synchronous vs. asynchronous computation in VLSI array processors.
In *Proceedings of SPIE Symposium, Vol. 341, Real-Time Signal Processing V*. The Society of Photo-Optical Instrumentation Engineers, May, 1982.

32] Charles E. Leiserson.
Systolic priority queues.
Technical Report CMU-CS-79-115, Carnegie-Mellon University, Computer Science Department, April, 1979.

33] Charles E. Leiserson.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, Computer Science Department, October, 1981.

34] R. J. Lipton, S. C. Eisenstat and R. A. DeMillo.
Space and time hierarchies for classes of control structures and data structures.
Journal of the ACM 23(4):720-732, October, 1976.

35] Clifford G. Lob and Alexander O. Elkins.
18-Mhz clock distribution system.
Hewlett-Packard Journal, August, 1983.

36] F. J. MacWilliams and N. J. A. Sloane.
The Theory of Error-Correcting Codes.
North-Holland, Amsterdam, Holland, 1977.

37] Carver A. Mead and Lynn A. Conway.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Mass., 1980.

38] C. A. Mead and M. Rem.
Cost and performance of VLSI computing structures.
IEEE Journal of Solid State Circuits SC-14(2):455-462, April, 1979.

39] Robert C. Minnick.
A survey of microcellular research.
Journal of the Association for Computing Machinery 14(2):203-241, April, 1967.

40] Thomas A. Ottmann, Arnold L. Rosenberg and Larry J. Stockmeyer.
A dictionary machine (for VLSI).
IEEE Transactions on Computers C-31(9):892-897, September, 1982.

41] M. S. Paterson, W. L. Ruzzo and L. Snyder.
Bounds on minimax edge length for complete binary trees.
In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 293-299. ACM SIGACT, May, 1981.

42] W. W. Peterson and E. J. Weldon, Jr.
Error-Correcting Codes.
MIT Press, Cambridge, Massachusetts, 1972.

43] I. V. Ramakrishnan and Peter J. Varman.
Modular matrix multiplication on a linear array.
IEEE Transactions on Computers C-33(11):952-958, November, 1984.

44] C. L. Seitz.
System Timing.
Chapter 7 of Mead and Conway [37].

45] Arun K. Somani and Vinod K. Agarwal.
An efficient VLSI dictionary machine.
In *Proceedings of the Eleventh International Symposium on Computer Architecture*. June, 1984.

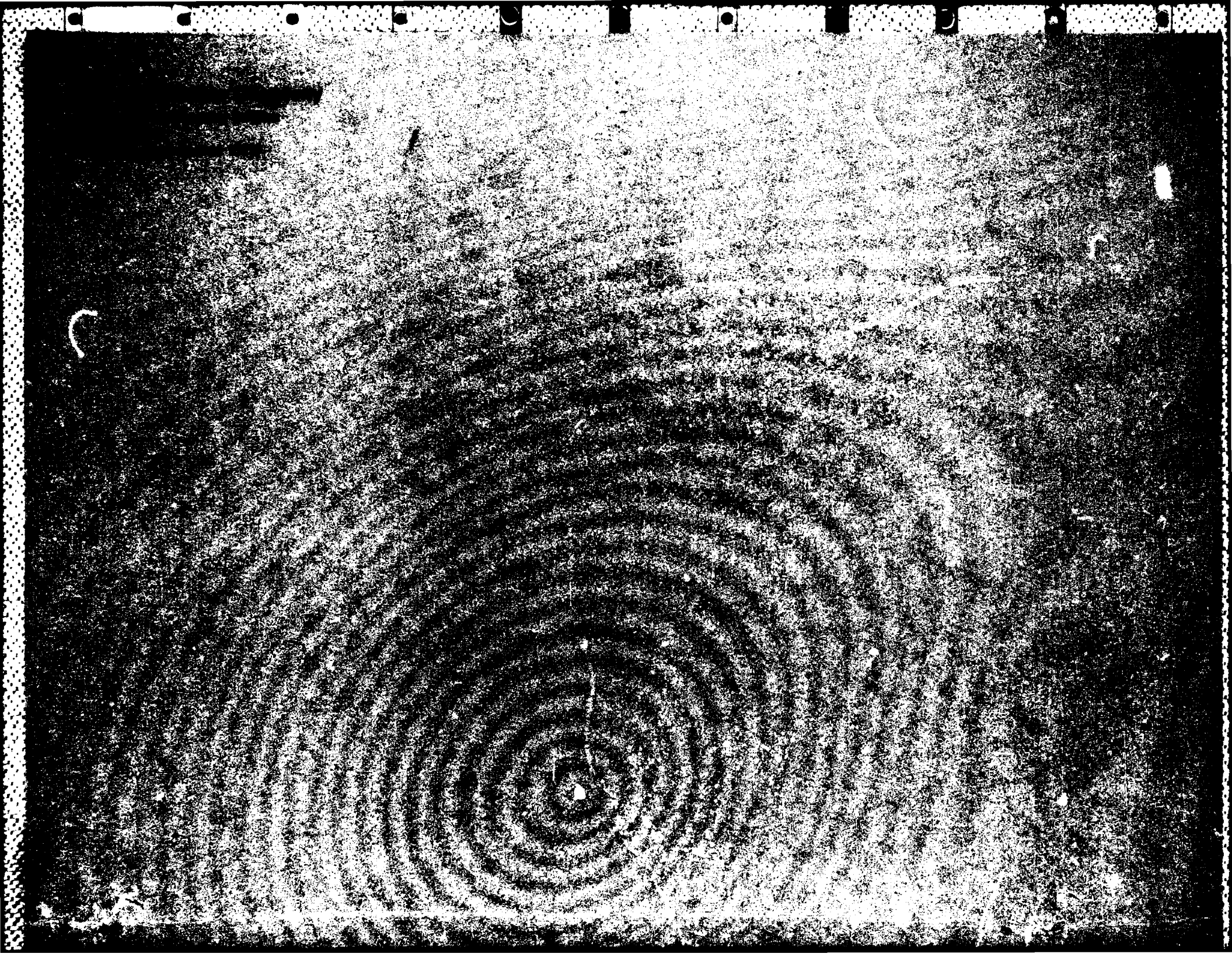
46] E. E. Swartzlander, Jr., B. K. Gilbert and I. S. Reed.
Inner product computers.
IEEE Transactions on Computers C-27(1):21-31, 1978.

47] J. J. Symanski.
A systolic array processor implementation.
In *Proceedings of SPIE Symposium, Vol. 298. Real-Time Signal Processing IV*. The Society of Photo-Optical Instrumentation Engineers, August, 1981.

48] J. J. Symanski.
Progress on a systolic processor implementation.
In *Proceedings of SPIE Symposium, Vol. 341. Real-Time Signal Processing V*, pages 2-7. The Society of Photo-Optical Instrumentation Engineers, May, 1982.

49] C. D. Thompson.
A Complexity Theory for VLSI.
PhD thesis, Carnegie-Mellon University, Computer Science Department, August, 1980.

50] D. W. L. Yen and A. V. Kulkarni.
Systolic processing and an implementation for signal and image processing.
IEEE Transactions on Computers C-31(10):1000-1009, October, 1982.



END

FILMED

8-85

DTIC